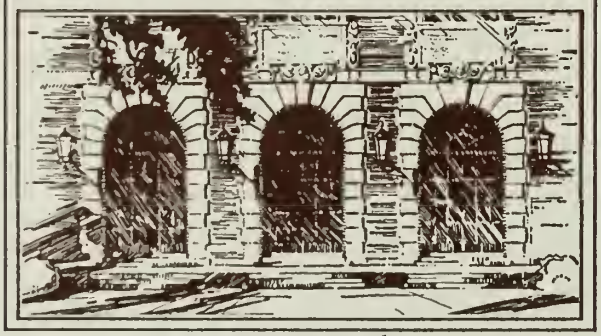


Report No. 364

A MACRO-ASSEMBLER FOR ILLIAC IV

by

David Michael Grothe

December 1, 1969

DEPARTMENT OF COMPUTER SCIENCE
UNIVERSITY OF ILLINOIS AT URBANA-CHAMPAIGN · URBANA, ILLINOIS

Report No. 364

A MACRO-ASSEMBLER FOR ILLIAC IV*

by

David Michael Grothe

December 1, 1969

Department of Computer Science
University of Illinois at Urbana-Champaign
Urbana, Illinois 61801

* This work was supported in part by the Advanced Research Projects Agency as administered by the Rome Air Development Center under Contract No. USAF 30(602)-4144 and submitted in partial fulfillment of the requirements for the degree of Master of Science in Computer Science, September 1969.

ABSTRACT

This report describes the ILLIAC IV macro assembly language (ASK) and the ILLIAC IV macro assembler. ASK is a free field assembly language with conditional assembly features and in-line text-substitution macros.

ACKNOWLEDGMENT

The author wishes to express his gratitude to N. Saville without whose encouragement (and arguments) the ILLIAC IV macro assembler would not have achieved its present and proposed states.

The author also wishes to thank Professor R. S. Northcote for his support and encouragement in this project.

Also, the author is indebted to the ILLIAC IV project for providing the opportunity to create this assembler.

Finally, the author would like to thank Mrs. Kay Flessner and Mrs. Patricia Douglas for the typing of this manuscript.

TABLE OF CONTENTS

Page

Page

LIST OF FIGURES

PREFACE

This paper deals with the design and implementation of a macro-assembler for ILLIAC IV.

Chapter 1 discusses the features of the ILLIAC IV hardware which pose assembler design problems, such as addressing and alignment of program and data.

Chapter 2 discusses briefly the operating system environment of the assembly program.

Chapter 3 defines the ASK language as it exists at the time of writing.

Chapter 4 treats the macro, and conditional assembly features of ASK and discusses their implementation.

1. ILLIAC IV ARCHITECTURE AND ADDRESSING

1.1 ILLIAC IV Architecture*

1.1.1 Processors

ILLIAC IV separates the functions of controlling the hardware and processing of "data manipulating" instructions into two logically distinct, but interacting, machines. A control unit (CU) exists which is itself programmable. It has its own repertoire of instructions which enable it to control both itself and the remainder of the hardware. The hardware external to itself, which the CU controls, is an array of sixty-four processing elements (PE's). The PE's execute instructions, passed to them by the CU, in lockstep. Figure 1 illustrates this relationship pictorially. ASK (Assembler System-K) must assemble, into a single stream of instructions, instructions for both these machines.

1.1.2 Memory

Both the CU and the PE's have associated memory. The CU contains an assortment of control registers (Instruction Counter, Interrupt Register, etc.), data registers (sixty-four 64-bit temporary storage registers and four 64-bit Index/Utility Registers). Each of these registers is assigned a fixed location in CU-Memory. That these registers are addressed by the hardware as CU-Memory locations suggests that ASK provide for symbolic addressing of CU-Memory (ASK does in fact provide this facility).

Each PE has associated with it a memory stack of its own (2048 64-bit words). A PE may address only its own memory stack, but the entire array of PE-Memory may be addressed by the CU (64 times 2048 = 2^{17} 64-bit words). Program is stored in PE-Memory and is fetched into the CU for execution.

---

*The ILLIAC IV computer is described by Barnes, et. al. in [5] and defined in detail in [3].

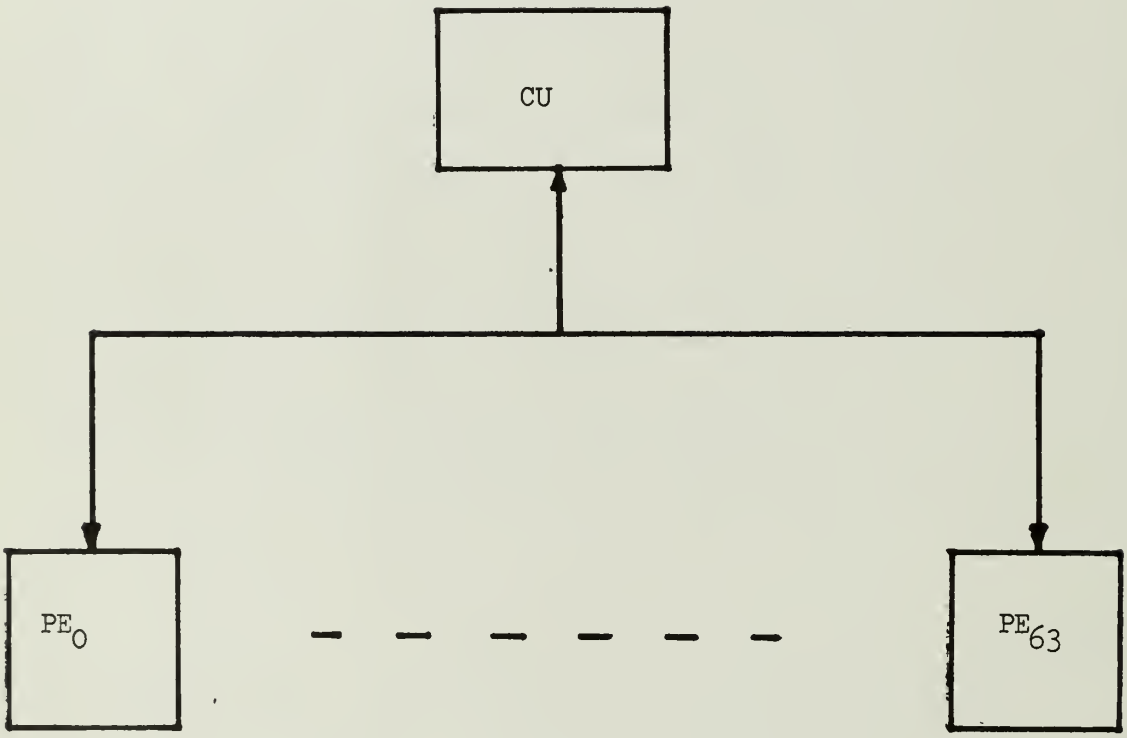


Figure 1. Each Square Represents a Processor, 65 Total.

The addressability of PE-Memory presents a problem for symbolic assembly: The instruction counter addresses PE-Memory as if it were a string of 2^{18} 32-bit syllables; the CU index registers may address PE-Memory as if it were 2^{17} 64-bit words; and since all PE's receive the same address from the CU, except for PE indexing, the PE's address PE-Memory as if it were 2^{11} 4096-bit words (4096 (= 64 times 64) is regarded as the word size since sixty-four PE's each fetch a 64-bit word from each PE memory simultaneously). (See Figure 2.) The problem arises from the possibility of a single symbolic address being used in all three contexts. Since it is desirable that the symbol denote a unique location in PE-Memory, a definition of assembly-time arithmetic (address arithmetic) must be chosen which satisfies this requirement.

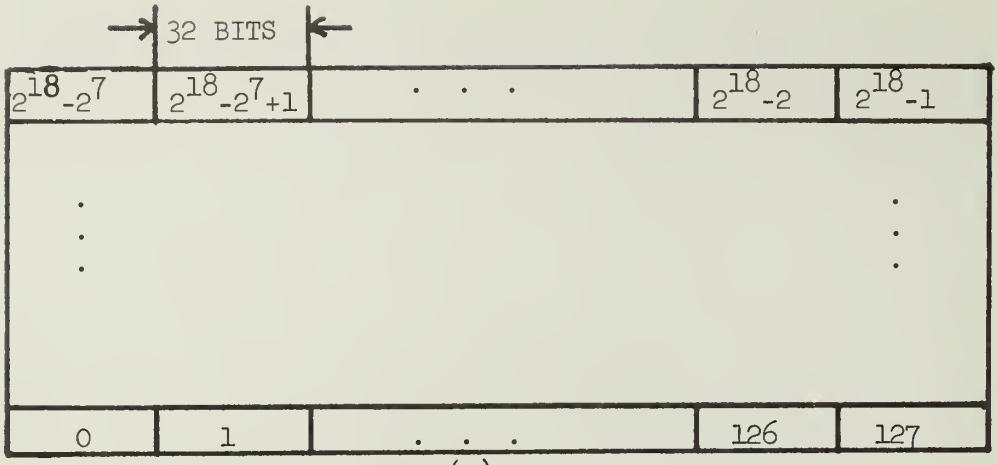
1.2 Assembly-Time Arithmetic

1.2.1 Three Modes of Arithmetic: Syllable, Word, Row

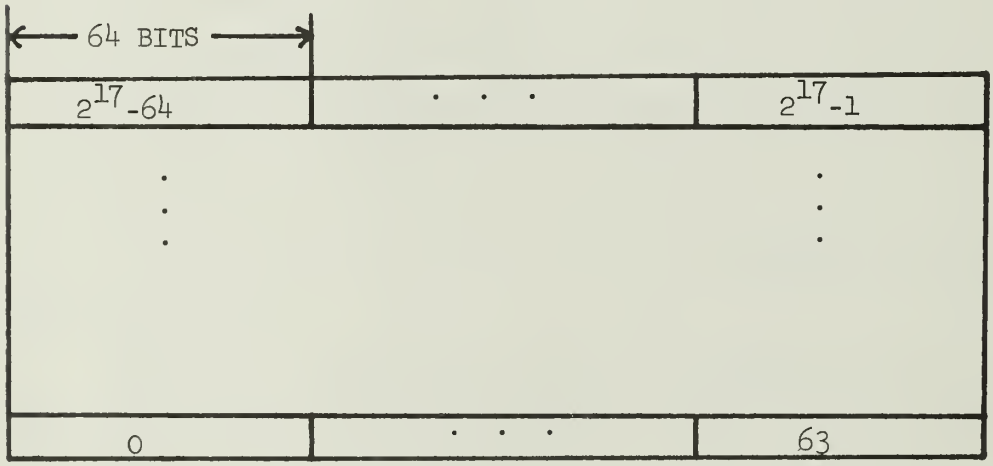
ASK evaluates arithmetic expressions using one of three modes of arithmetic, depending upon context. Syllable arithmetic operates on a PE symbol* as if it symbolizes the PE memory address of a 32-bit instruction syllable; word arithmetic operates on a PE symbol as if it symbolizes the PE memory address of a 64-bit word; row arithmetic operates on a PE symbol as if it symbolizes the PE memory stack address of an entire row of 64-bit words across a quadrant. Since the same PE symbol may, at different times, appear in all three contexts, it would not be meaningful to use the same value for the symbol in each of the three modes of arithmetic.

For instance, a PE symbol PLACE which has the value 23 would represent three entirely different memory locations if the number 23 were used as a

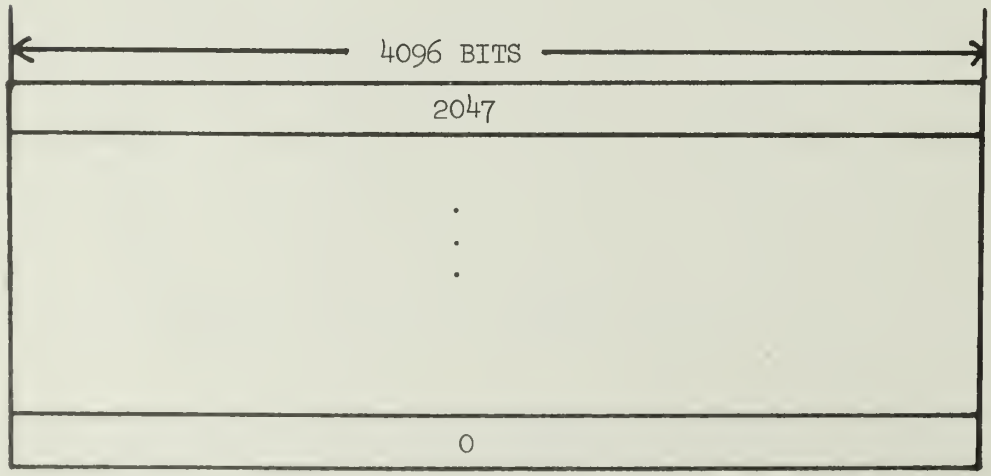
*Section 3.2.2 gives a syntactic description of <PE symbol>.



(a)



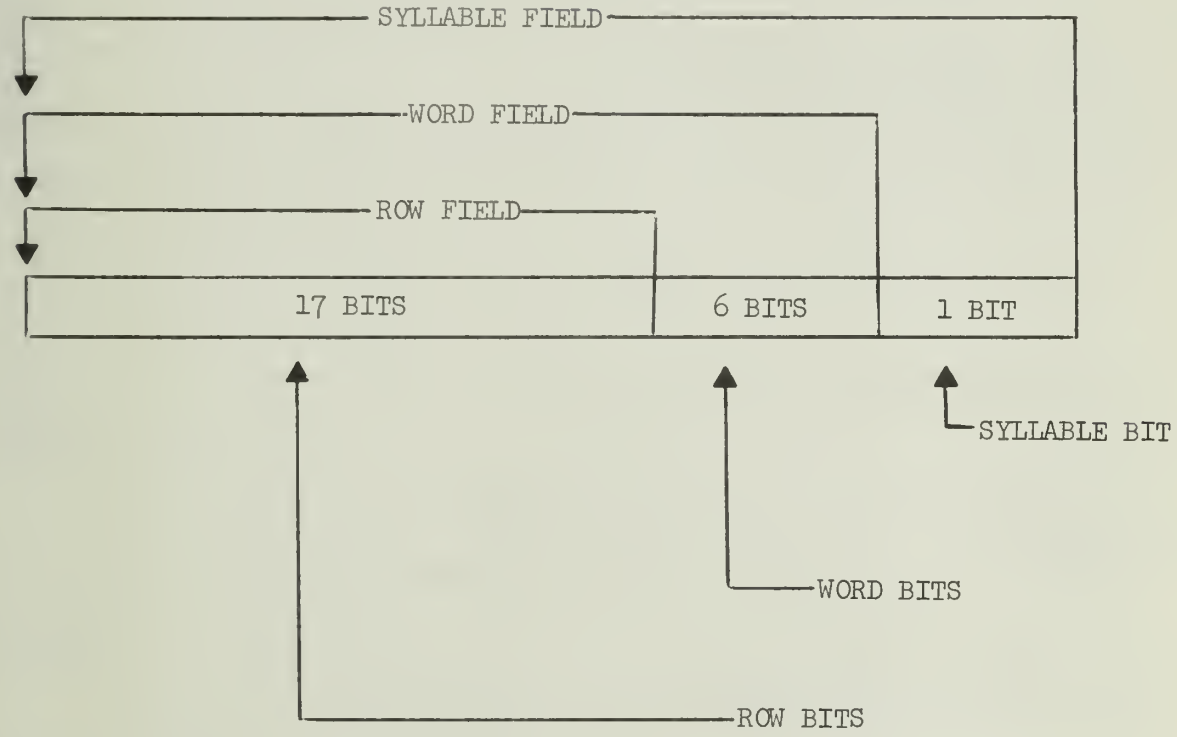
(b)



(c)

Figure 2. PE Memory As Seen By (a) Instruction Counter, (b) CU Address Registers, (c) PE Address Logic.

syllable address, word address, and row address. In order to avoid this ambiguity, ASK considers the value of a PE symbol to be divided into three fields for purposes of evaluating arithmetic expressions.



The above diagram represents the value of a PE symbol as it is interpreted by ASK. Syllable arithmetic operates on the syllable field; word arithmetic operates on the word field; row arithmetic operates on the row field.

The interpretation of a numeric value depends upon how that value was specified in the source text:

1) The value of a PE symbol is interpreted as specified in the preceding paragraph.

2) The value of a CU symbol* or a numeric constant is interpreted as designating the same field as the mode

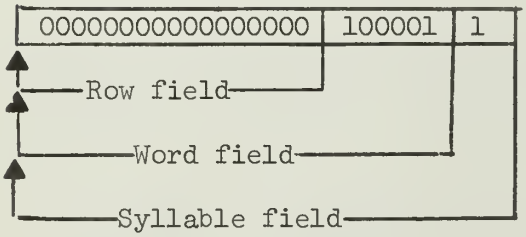
---

*See Section 3.2.2 for a definition of the syntactic item <CU symbol>

of arithmetic being performed on it. For example, the numeric constant 23 designates syllable, word and row 23 in syllable, word and row arithmetic, respectively.

Examples:

Suppose the PE symbol A has the value 67_{10}.



Syllable Arithmetic	A+2 = 69
Word Arithmetic	A+2 = 33
Row Arithmetic	A+2 = 2

1.2.2 Arithmetic Expressions

Assembly-time arithmetic expressions will now be defined syntactically and semantically:

Syntax:

<arithmetic expression> ::= <term>|<adding operator> <term>|
<arithmetic expression> <adding operator> <term>

<term> ::= <factor>|<term> <multiplying operator> <factor>

<adding operator> ::= + | -

<multiplying operator> ::= × | /

<factor> ::= <arithmetic primary>|<factor>
<exponentiation operator> <arithmetic primary>

<arithmetic primary> ::= (<arithmetic expression>)|<integer>|
<symbol>|<allocation counter designator>|
ABS (<arithmetic expression>)|
RELOC (<arithmetic expression>)|
SLA (<arithmetic expression>)|
WDA (<arithmetic expression>)|
RWA (<arithmetic expression>)

<allocation counter designator> ::= <space> @@ <space>

<exponentiation operator> ::= *

<space> ::= {one or more consecutive blank characters}

Semantics:

An arithmetic expression denotes a sequence of arithmetic operations to be performed (at assembly time) on certain specified quantities. The operations allowed are: addition, subtraction, multiplication, integer division and exponentiation (raised to the power of). Evaluation is performed in 24-bit two's complement arithmetic.

ABS specifies that the result of the evaluation of the arithmetic expression is to be made absolute (no matter what the relocatability of the expression turns out to be).

RELOC acts the same as ABS only the value is made relocatable.

SLA indicates that the parenthesized expression is to be evaluated using syllable arithmetic.

WDA indicates that the parenthesized expression is to be evaluated using word arithmetic.

RWA indicates that the parenthesized expression is to be evaluated using row arithmetic.

Examples

1

(3)

X + 3

PLACEINMEMORY + Y/2*(X-1) + 2 × N

1.2.3 Relocatable Arithmetic

During the assembly of any particular code segment, it may not be known where in PE memory the object code will actually be loaded. Therefore, ASK must make provision as it emits "object" code, for the placement of that

code at an "arbitrary" place in PE memory.* An "object" code file with that property is known as a relocatable code file. The assembly proceeds as if the code were to be loaded at PE memory location zero. At load time, however, the code may be loaded at PE memory address R. Therefore, if a PE symbol symbolizes location m at assembly time, it must symbolize location R+m at load time. Relocatable arithmetic takes the term R into account during the evaluation of arithmetic expressions.

In the following analyses:

Let R_s and R_s^1 stand for PE symbols which symbolize some PE memory address which may be relocated.

Let A_s and A_s^1 stand for either an integer or a symbol which symbolizes a PE memory address which may not be relocated. Henceforth a quantity of one of these two types shall be referred to as an absolute quantity.

Let m and n stand for the numbers associated with the symbols, and R stand for the starting PE memory address of the code at load time.

ADDITION

Three cases:

(1) $R_s + R_s^1 = (R+m) + (R+n)$

$= 2R + (m+n)$

This result is valid only for intermediate results. An expression which evaluates to a relocation amount greater than R is invalid and is flagged as such at assembly time.

(2) $R_s + A_s = (R+m) + (n)$

$= R + (m+n)$

---

*Section 1.3 places some restrictions on how arbitrary "arbitrary" can be.

This result is valid under all circumstances which allow a relocatable expression. The assembly time result is (m+n) as a relocatable quantity.

(3) $A_s + A_s^1 = m + n$

This result is the number (m+n) which is absolute (not relocatable) and as such is valid under any circumstances which allow absolute quantities.

SUBTRACTION

Four cases:

(1) $R_s - R_s = (R+m) - (R+n)$

$= (R-R) + (m-n)$

$= m - n$

The result of subtracting two relocatable quantities is an absolute quantity.

(2) $R_s - A_s = (R+m) - n$

$= R + (m-n)$

The result of subtracting an absolute quantity from a relocatable one is a relocatable quantity (m-n).

(3) $A_s - R_s = n - (R=m)$

$= (n-m) - R$

This result produces a negative relocation amount which is invalid.

(4) $A_s - A_s = m-n$

The result of subtracting one absolute quantity from another one is their difference (m-n), which is also absolute.

MULTIPLICATION

Three cases:

(1) $R_s \times R_s = (R+m) \times (R+n)$

$= R^2 + R \times m + R \times n + m \times n$

Multiplication of two relocatable quantities is invalid under all circumstances.

(2) $R_s \times A_s = (R+m) \times n = (R \times n) + (m \times n)$

Multiplication of a relocatable quantity and an absolute quantity is invalid under all circumstances.

(3) $A_s \times A_s = m \times n$

The only valid multiplication is that of two absolute quantities.

INTEGER DIVISION (All address arithmetic is integer arithmetic)

Four cases:

(1) $R_s / R_s = (R+m) / (R+n) = R / (R+n) + m / (R+n)$

Division of one relocatable quantity by another is invalid under all circumstances.

(2) $R_s / A_s = (R+m) / n = R/n + m/n$

Division of a relocatable quantity by an absolute quantity is invalid under all circumstances.

(3) $A_s / R_s = n / (R+m)$

Division of an absolute quantity by a relocatable quantity is invalid under all circumstances.

(4) $A_s / A_s = m/n$

The only valid division is that of two absolute quantities.

EXPONENTIATION

Exponentiation is multiplicative in nature and obeys the same rules as multiplication. The only valid construct is:

$A_s * A_s^1 = m^n$

Summary:

The valid constructs in relocatable arithmetic are:

$R_s + R_s$	Valid only as an intermediate result.
$R_s + A_s$	Relocatable.
$A_s + A_s$	Absolute.
$R_s - R_s$	Absolute.
$R_s - A_s$	Relocatable.
$A_s \times A_s$	Absolute.
A_s / A_s	Absolute.
$A_s * A_s$	Absolute.

An arithmetic expression is correct, with respect to relocatability, if the final result contains either the term $1 \times R$ (as in R_s) or $0 \times R$ (as in A_s). A further contextual restriction may be applied where only an absolute or only a relocatable result is valid.

Examples of Relocatable Arithmetic:

Let a symbol which begins with the letter "R" be understood to be relocatable, and one which begins with the letter "A" be understood to be absolute.

RX + RY - RA	Relocatable.
RX - (AY + RA)	Absolute.
(RY - RX)/2	Absolute.
RX + RY	Invalid.
2 × RX	Invalid.
RX/2	Invalid.

1.2.4 External References

An address expression may make use of a symbol whose definition is external to the assembly in which it appears. This facility allows the possibility of total separation of code and data in PE memory, since a data area can be "declared" via a single (or collection of) ASK program(s) consisting entirely of data-loading and/or storage reservation pseudo instructions. These data areas may then be addressed by another ASK program consisting entirely of executable code. The fact that the value of an external symbol is not known at assembly-time leads to the necessity that certain restrictions must be placed on the use of such symbols:

(1) An external symbol may not appear in any expression whose result determines, in any way, the size of the program, i.e., storage reservation pseudo operations.

(2) An external symbol may not appear as a factor in an expression in conjunction with a multiplicative operator.

(3) An external symbol may appear as a term in an expression in conjunction with an additive operator only if the other terms of the expression (considering the external symbol as having value 0, Absolute) comprise an absolute expression.

(4) (From (3)) At most one external symbol may appear in a single expression.

Thus, an external symbol or expression represents a base in PE-Memory (unknown at assembly-time) ± a value known absolutely at assembly-time.

A symbol is declared as external to ASK via the following pseudo-declarations:

<External Declaration> ::= EXTERNAL <External Symbol List>

<External Symbol List> ::= <External Symbol> | <External Symbol List> ,
<External Symbol>

<External Symbol> ::= <PE Symbol>

The user may indicate to ASK that a symbol which is defined within this assembly is to be made visible externally. A symbol so defined may correspond to the same symbol declared EXTERNAL in another assembly and, if it does correspond, give the EXTERNAL symbol a definition at load time. Such symbols are termed Entry Symbols in ASK and are defined as such by way of the following syntax (an extension of the syntax for <Instruction Label> given in Section 3.2.4):

<Instruction Label> ::= <PE Symbol> | <PE Symbol> [ENTRY]

1.3 Boundary Considerations

A good portion of the art of programming the ILLIAC IV lies in the structuring of the data to be operated on. In particular, in order to take advantage of certain functional characteristics of the hardware, it is sometimes convenient -- and sometimes necessary -- that code or data be placed at some specific address with respect to some address whose value is a multiple of some particular power of two.

For example:

a) There exists an instruction which fetches eight words of data from PE-Memory to CU-Memory. This instruction demands that the data be on an eight-word bounday.

b) The JUMP instruction requires that the syllable to be jumped to lie on a single-word boundary.

c) PE instructions, unless modified by the PE index registers, fetch data from the same PE-Memory address, implying that the data should be placed beginning on a sixty-four-word boundary (or 128 or 256, depending upon the configuration).

The above suggests the need for a facility which allows a PE Symbol to be assigned a value, v, such that $v \equiv w$ modulo t, where t is a power of two. If t is determined implicitly from w, i.e., the smallest power of two greater than or equal to w, then w determines the congruence class $V = \{v \mid v \equiv w \text{ modulo } t\}$. Accordingly, ASK provides a facility through which the Allocation Counter (the assembly-time image of the Instruction Counter) may be advanced to the nearest v such that $v \in V$, given w. This allows code or data to be placed at any of these convenient boundaries.

This, however, is not in itself sufficient for, although the address v may appear at assembly time to satisfy the constraints, it is not necessarily known whether the actual run-time address will or not. The solution to this problem is that the ILLIAC IV loader is informed of the existence of such constraints. The manner in which the loader is informed of these matters is crucial. Suppose, for instance, that ASK simply provided the loader with the constant w at each occurrence of such an address adjustment. Were this the case, the loading address of a code segment possibly varying from run to run, the loaded code segment could be considerably larger (or smaller) than it was thought to be at assembly time, invalidating many relocatable addresses.

A second, acceptable, approach is this:

For every code segment (assembly) there exists a number T which represents the largest t_i for that segment. If the loading address (L) is such that $L \equiv 0$ modulo T, then for all v_i such that $v_i \equiv w_i$ modulo t_i, it follows that $L + v_i \equiv w_i$ modulo t_i. That is to say, the actual memory

satisfies the constraints imposed upon it at assembly time, and no adjustment must be performed except prior to determining L. Hence, the code segment remains the same size as it was thought to be at assembly time, and all relocatable addresses are valid.

The latter approach to the "Boundary Problem" was the one selected for implementation. (See Section 3.2.8.5 - FILL Pseudo).

2. OPERATING SYSTEM ENVIRONMENT

"Die Welt ist alles, was der Fall ist." -- Ludwig Wittgenstein, 1.

ASK must interface to (at least) three separate, but connected, operating systems.

The first of these is OS4, the ILLIAC IV resident operating system. ASK does not itself run under the auspices of OS4, but the code which it emits, when loaded into ILLIAC IV and executed, does. At the time of this writing, OS4 is in its first, infantile state. The conventions established by this operating system are minimal and, to some extent, arbitrary. It is envisioned that, in the future, OS4 will expand as more user facilities are necessitated by the needs of an increasingly large and (hopefully) diverse world of users. ASK must provide for the interface to this operating system, taking into account the fact that the system itself is likely to change. The only assumption that ASK can make is that the language for communicating with OS4 is defined, namely ILLIAC IV machine language. ASK will provide this interface not via special constructs in its own language but, rather, by way of macros written by the operating system group, known to ASK as macros and thus available to the user. This approach (the "System Macro" approach) obviates the necessity of changing ASK itself as OS4 changes; the only necessary change is to the definitions of the "System Macros" which are input to ASK.

The second operating system that ASK must interface to is the B6500 Resident ILLIAC IV Operating System (this operating system has no official name at this time, but will herein be called BRIOS). The design of this operating system makes this interface simple almost to the point of non-existence. To BRIOS, ASK is just a B6500 program which it runs at appropriate times. The interface to BRIOS consists mainly of not interfering with it, specifically by not initiating any other B6500 jobs whose supervision belongs

to BRIOS. The only other possible interface to BRIOS is the returning of a condition code to indicate the degree of success of the assembly.

The third operating system which ASK interacts with is the B6500 MCP. The consciousness of the MCP affects little other than the manner in which ASK itself is coded. Although these considerations are important, they will not be discussed here.

There is a fourth system to which ASK must interface very closely, although it is not an operating system; that is, the ILLIAC IV Collector/Loader system. The closeness of this interface very nearly binds the two systems into a single unit. Any change to one system which affects this interface could conceivably imply extensive modifications to the other. The reader is referred to [1] which defines this interface in detail.

3. THE ASK LANGUAGE

This section defines the ASK language in detail. The description is taken, in part, from [2], a reference manual for ASK. A familiarity with ILLIAC IV and its instruction set [3] is assumed for total comprehension of this section.

3.1 General Format of Input to ASK

ASK accepts as input punched cards or card images from any source available to the B5500 (B6500) system. The information in card (image) columns 1-72, inclusive, is considered by ASK to be statements or portions thereof in the ASK language. Columns 73-80 are available to the user for sequencing or identification purposes except when ASK is performing a merge assembly from two sources, in which case the information in columns 73-80 are used by the selection criterion to determine the next card to be scanned (see Section 3.2.1 ASK Control Statements). An arbitrary number of blank spaces may separate any two syntactic quantities and, with the following exceptions, card (image) boundaries are ignored:

a) Neither an identifier nor a number may be split across a card boundary or contain embedded blanks.

b) A "$"-sign in column one followed by one or more spaces will cause the card to be interpreted as the beginning of an ASK control statement.

3.2 Syntactic and Semantic Description of ASK

3.2.1 ASK Control Statements

Control statements direct the assembler to take some action, usually with respect to file handling. A control statement may appear anywhere in an

ASK program, allowing assembly-time options to be manipulated in accordance with the desires of the user.

The following is a syntactic and semantic description of ASK control statements:

Syntax

<ASK control statement> ::= $ <verb list>

<verb list> ::= <verb> | <verb list> <verb>

<verb> ::= <input specifier> |
<output specifier> |
<patch specifier> |
<option specifier>

<input specifier> ::= <input file designator> <label equation>

<input file designator> ::= CARD | TAPE1 | TAPE2 | TAPE3 | TAPE4 |
TAPE5 | TAPE6 | TAPE7 | TAPE8 |
TAPE9 | TAPE10 | TAPE11 | TAPE12 |
TAPE13 | TAPE14 | TAPE15

<label equation> ::= <empty> |
= <multi-file id>/<file id> <disk or tape file>

<disk or tape file> ::= SERIAL | <empty>

<multi-file id> ::= <identifier>

<file identifier> ::= <identifier>

<output specifier> ::= <output file designator> <label equation>

<output file designator> ::= NEWDISK | NEWTAPE

<patch specifier> ::= MERGE <label equation> |
VOID <base ten number>

<option specifier> ::= LIST | SYNTAX |
XREF | BLOWUP | PUNCH |
SEQ | SEQ + <base ten number>

Semantics

A <control statement> causes the assembler to change its mode of operation with respect to file handling or listing options.

An <input specifier> directs ASK to accept symbolic input from a file of the user's choice. The file CARD is the main input file for ASK, i.e., ASK must find its first input in file CARD. If ASK is directed to another input file, it assembles from that file until either it encounters a control card with an input specifier or reads the end of file marker. In the former case, ASK begins assembling from the new file and "remembers" which file it was assembling from. In the latter case, ASK closes the file from which the EOF was read and continues assembling from the file which contained the control statement which directed it to the file it has just closed. Assembly proceeds from the card image immediately following the control statement in this case. If the <label equation> part is non-empty, ASK attaches itself to the specified tape or disk file.

If more than one <input specifier> is given in an <ASK control statement>, ASK will assemble from the file which is listed last until an EOF is reached. Then it will assemble from the file listed next to last until an EOF is reached. ASK will continue in this fashion until all input files listed in the control statement are exhausted. It will then go back to assembling the file in which the <ASK control statement> appeared.

An <output specifier> directs the assembler to create a new symbolic tape or disk file. This file will contain the totality of card images which ASK has processed from whatever files their origin may have been. Once an output specifier has been used, it is not necessary to specify it on subsequent control cards, since the option remains on for the rest of the assembly. It is possible, however, to direct ASK to create different output files for different sections of code by placing several control statements with an output specifier and label equation in the source file.

If the <patch specifier> is used, ASK considers the totality of card images from the files available as input file designators as an update deck for

file MERGE. The functions of replacement, deletion and insertion are available. The selection criterion for which card image ASK will next process is the sequence number comparison between the next available card image from file MERGE and the designated input file. The selection algorithm is as follows:

	Relation between Sequence Numbers	File from Which Input is Taken
1)	"PATCH" sequence < MERGE sequence	"patch"
2)	"PATCH" sequence = MERGE sequence	"patch"
3)	"PATCH" sequence > MERGE sequence	MERGE

In case 1), the card from the MERGE file is retained for subsequent comparisons. In case 2), the card from the MERGE file is discarded so that the next card from that file can be used for the next comparison. In case 3), the card from the "patch" file is retained for subsequent comparisons.

If the VOID option is used, ASK discards card images from file MERGE as long as the sequence number from card images in file MERGE remains less than or equal to the value of the <base ten number>. The VOID is performed when its sequence number is less than or equal to the sequence number of the next card from file MERGE. Once ASK begins merging it continues to do so until the assembly is terminated or an EOF is read from file MERGE, at which point the user may choose to complete the assembly from the "patch" file or attach ASK to another file MERGE. The user may at any time attach ASK to another file MERGE through the use of the label equation construct.

At the time that a control statement is encountered, each of the options which may be an <option specifier>, except SEQ, is set to FALSE. The presence of the option specifier verb enables that particular option. The options and their effects are as follows:

LIST — The source program and instructions being generated are listed on the printer file.

SYNTAX — The generated object code is inhibited from being written into the object code file.

XREF — ASK is to cross-reference all identifiers, register designators, and control verbs as they are encountered and print out the cross reference table at the end of the assembly.

BLOWUP — When printing the generated instructions, ASK will print all ILLIAC IV instructions in an "exploded view" with each field of the instruction displayed individually, in octal, separated from neighboring fields by a single space.

PUNCH — Causes ASK to punch each card image as it is processed. "Dollar cards", <ASK control statements>, are not punched.

SEQ — Causes ASK to resequence whatever source code output it is creating. The sequence increment is set equal to the value of the arithmetic term (evaluated using word arithmetic). If no term is given, a default value of 100 is used.

Examples:

Control statement:

```
$ LIST SYNTAX XREF
  NEWDISK = SOURCE/CODE SERIAL
  SEQ + 1000
$ TAPE1 = SINE/ROUTINE SERIAL
  LIST  XREF
$ LIST PUNCH SEQ + 10000
$ NEWTAPE
$ LIST VOID 19300 SYNTAX
```

$ TAPE1 = LAST/DONE TAPE5 = THIRD/DONE

TAPE3 = SECOND/DONE TAPE12 = FIRST/DONE

3.2.2 Basic Elements of the Language

3.2.2.1 Characters and Identifiers

Syntax:

<character> ::= A|B|C|D|E|F|G|H|I|J|K|L|M|N|O|P|Q|R|S|T|U|V|W|X|Y|Z|
0|1|2|3|4|5|6|7|8|9|.|[|(|<|←|$|*|)|;|≤|-|/|,|%|=|
]|#| |@|:|>|≥|+|×|≠|?|"

<letter> ::= A|B|C|D|E|F|G|H|I|J|K|L|M|N|O|P|Q|R|S|T|U|V|W|X|Y|Z

<numeric character> ::= 0|1|2|3|4|5|6|7|8|9

<alphanumeric character> ::= <letter>|<numeric character>

Semantics:

The character set for the assembly language for ILLIAC IV is the 6-bit character set which exists on the Burroughs B5500. An identifier may symbolize things such as a machine instruction, an address in PE memory, or a number. An identifier is restricted to be no more than 63 characters in length.

3.2.2.2 Symbols

Symbols in ASK provide a mnemonic means of designating such entities as PE-Memory locations, CU-Memory locations, and ILLIAC IV registers.

Syntax:

<PE symbol> ::= <identifier>

<CU symbol> ::= .<identifier>

<register symbol> ::= $<identifier>

<symbol> ::= <PE symbol>|<CU symbol>

<identifier> ::= <letter>|<identifier> <alphanumeric character>

Semantics:

Although a PE symbol may symbolize an address in PE memory, its semantic interpretation is not restricted to that alone. A PE symbol is best interpreted as symbolizing a number, with the understanding that this number itself takes on quite different meanings depending upon the context in which it is used. ASK attaches no meaning (other than its numeric value) to a symbol at the time it is defined.

A PE symbol may have a numeric value of up to 24 bits of precision.

A CU symbol may symbolize an address in CU memory. All the remarks about semantic interpretation of PE symbols apply to CU symbols as well. A CU symbol is restricted to 62 alphanumeric characters in length (+ 1 for the . = 63) and it may assume a value of no greater than 8 bits of precision. If a CU symbol is defined by a quantity of greater precision than 8 bits, the quantity is truncated to 8 bits of precision.

A register symbol denotes an ILLIAC IV hardware register. Certain register symbols are predefined by ASK (see sections 3.2.5 and 3.2.6), but the user is at liberty to define other register symbols from existing ones. A register symbol may be used only in a context which calls for or allows reference to an ILLIAC IV hardware register.

3.2.2.3 Numbers

Syntax:

<integer> ::= <integer part> <base specifier>

<integer part> ::= <base ten digit> | <integer part> <digit>

<base specifier> ::= :<base ten number> | <empty>

<base ten number> ::= <base ten digit> | <base ten number>
<base ten digit>

<base ten digit> ::= 0|1|2|3|4|5|6|7|8|9

<digit> ::= 0|1|2|3|4|5|6|7|8|9|A|B|C|D|E|F|G|H|I|J|K|L|M|N|O|P|Q|R|
S|T|U|V|W|X|Y|Z

<real number> ::= <signed real number> | <unsigned real number>

<signed real number> ::= +<unsigned real number> |
-<unsigned real number>

<unsigned real number> ::= <mantissa part> | <exponent part> |
<mantissa part> <exponent part> |
<base ten number> <exponent part>

<mantissa part> ::= <base ten number>. |
<base ten number>.<base ten number> |
.<base ten number>

<exponent part> ::= @<signed base ten number> | @<base ten number>

<signed base ten number> ::= +<base ten number> | -<base ten number>

<paired number> ::= PAIR (<real number or integer>,
<real number of integer>)

<real number or integer> ::= <real number> | <integer>

<number> ::= <integer> | <real number> | <paired number>

Semantics:

A number denotes its value. Integers are represented in fixed point binary with the binary point at the right. Real numbers are represented in ILLIAC IV floating point form (see page 3.3 on data formats [3] for details).

A digit must be such that its assigned weight is less than the specified base (or ten if the base is unspecified). The weights assigned to the possible digits are as follows:

digit:	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P
											Q	R	S	T	U	V	W	X	Y	Z						
weight:	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25
											26	27	28	29	30	31	32	33	34	35						

The base specifier directs the assembler to convert the preceding integer from the specified base to binary. If no base is specified, base ten is assumed.

A real number directs the assembler to perform conversion to 64-bit ILLIAC IV floating point representation. This conversion is performed if an explicit decimal point is present or if there is an explicit exponent part. In all other cases, integer conversion is performed.

The pair construct allows for the formation of two 32-bit words in inner-outer form. The first number is converted into the outer position, the second into the inner.

Examples:

```
0A:17
31
77332:8
@-8
.4@+37
PAIR (1.3@-1, 7765:8)
```

3.2.3 Structure of an ASK Program

Syntax:

```
<program> ::= BEGIN <compound statement>
                    <end statement>.
```

<end statement> ::= <labeled end statement> |
<unlabeled end statement>

<labeled end statement> ::= <label list> <unlabeled end statement>

<unlabeled end statement> ::= END |
END <arithmetic expression>

<compound statement> ::= <statement> |
<compound statement> ; <statement>

Semantics:

The <end statement> indicates the end of the assembly language program. The appearance of this mnemonic END causes a halt instruction to be generated. A jump instruction is generated after the halt. If no arithmetic expression is present, the jump is to relocatable location 0. If there is an arithmetic expression present, the jump is to the location indicated by the value of the arithmetic expression (evaluated using word arithmetic) with the same relocatability as the value of the expression. The arithmetic expression or the relocatable location 0 as the case may be, should be the location of the first instruction to be executed.

3.2.4 ASK Statements

Syntax:

<statement> ::= <ASK pseudo-op> |
<ASK control statement> |
<label list> <ILLIAC IV instruction> |
<ILLIAC IV instruction> | <empty>

<label list> ::= <instruction label> : |
<label list> <instruction label> :

<instruction label> ::= <PE symbol>

Note: the production for <ILLIAC IV instruction> is expanded in Appendix A. It includes all ILLIAC IV operation mnemonics and their allowable operands.

Semantics:

A statement in ASK denotes an operation to be performed, by either ILLIAC IV or the assembler, together with its necessary operands. A <label list> is limited to sixty-four or fewer consecutive <instruction labels>, and each is assigned the value of the Allocation Counter at the time of their encounter, i.e., the syllable address of the instruction being labeled.

3.2.5 Register Designators and Operand Fields for CU Instructions

CU OPERAND FIELDS

Syntax:

<compare and skip operand> ::= <CU memory address specifier> <ACARX> <skip field> <global-local specifier>

<CU memory operand> ::= <CU memory address specifier> <ACARX> <global-local specifier>

<skip operand> ::= <skip field> <global-local specifier>

<blank CU operand> ::= <global-local specifier>

<short literal operand> ::= <arithmetic expression> | = <arithmetic expression>

<long literal operand> ::= <number> | <symbol> | <index specifier>

<PE register specifier> ::= <PE register designator>

<mode bit specifier> ::= <mode bit>

<mode bit> ::= E|El|F|Fl|G|H|I|J

<ACAR selector> ::= (<arithmetic expression>)

<CU memory address specifier> ::= <arithmetic expression> | <CU register designator>

<skip field> ::= ,<arithmetic expression>

<global-local specifier> ::= ,G|,L|<empty>

<ACARX> ::= (<arithmetic expression>)| <empty>

<index specifier> ::= <arithmetic expression>, <arithmetic expression>,
<arithmetic expression>

Semantics:

Operand fields for CU instructions provide a symbolic method of determining the value of each field of the instruction syllable except the op-code fields.

A <blank CU operand> sets no fields except the global/local field.

A <short literal operand> sets the low order 24 bits of the instruction to the value of the arithmetic expression.

A <long literal operand> sets the next 64 bits (two instruction syllables) after the LIT instruction to the value of the number, symbol or index specifier.

A <PE register specifier> encodes a PE register in the address field of the instruction.

A <mode bit specifier> encodes a mode bit in the address of the instruction.

The <ACAR selector> sets the ACAR field of the instruction to the value of the arithmetic expression.

A <CU memory address specifier> sets the address field to the value of the arithmetic expression or to the CU memory address of the indicated register.

The <skip field> sets the skip field of the instruction to a value which is determined as follows:

The expression is evaluated using syllable arithmetic. If the result is relocatable, ASK sets the skip field to a displacement such that the destination of the skip is the instruction whose address is the value of the expression. That is, if the expression were simply L and L were relocatable, a skip to L would be generated by ASK. If the result is absolute, ASK uses that value as the skip distance itself.

The <global-local specifier> indicates that the instruction being generated is to be flagged as global (G), local (L), or in the same global-local mode as the "rest" of the program (see explanation of pseudos GLOBAL and LOCAL, sections 3.2.8.10 and 3.2.8.11).

The <ACARX>, if nonempty, sets the ACARX enable bit and bits 1:2 of the ACARX field to the value of the arithmetic expression modulo 4; otherwise, the ACARX field 0:3 is set to zero.

The <index specifier> indicates that 64 bits are to be set as three fields: bits 1:15, bits 16:24, and bits 40:24. These fields are set by the three arithmetic expressions respectively. Bit 0 of the 64 bits is not able to be set by this construct. In field one (bits 1:15), ASK forms a 15-bit sign-magnitude representation of the arithmetic expression. In fields two and three the 24-bit two's complement value is inserted as is.

With the exception of the <skip field>, all arithmetic expressions are evaluated using word arithmetic. With the exception of the <skip field>, <short literal operand>, and fields two and three of the <index specifier>, arithmetic expressions must have an absolute result. The above-mentioned exceptions may have either a relocatable or an absolute value.

Examples:

Compare and skip operand:

```
.DELTA, IOOP
$C3 , L+1
```

```
     .DELTA -1 (3) , LABEL, G
CU memory operand:
     $D34(2), L
     N x .STUFF  (PRESENTACARX)
     $TRI , L
Skip operand:
     ,LOOP
     ,L + 2
     ,- 14
     ,DESTINATION - ( @@ + 1 ),G
Blank operand:
     ,G
     ,L
Short literal operand:
     = SUBROUTINE
     = 77777777:8
      2*(N-1)
Long literal operand:
     = INCREMENT, LIMIT, INITIALVAL
     = -1, 0, 64
     = SCALEFACTOR
     = 1.7325@18
     = 10000000000000000000:8
     = -(2*14-1), -1, -1
ACAR selector:
     (REGISTER -1)
     (3)
     (2)
```

CU memory address specifier:

.LOCAL + 3 × (.Q-2*(N-1)+1)

$D2

$3D40

$C1

$ICR

$ACR

ACARX:

(XREGISTER -1)

(3)

(2)

REGISTER DESIGNATORS IN CU

Syntax:

<CU register designator> ::= $<quadrant specifier> <register mnemonic>|
$<register mnemonic>

<quadrant specifier> ::= 0|1|2|3

<register mnemonic> ::= D0|D1|D2|D3|D4|D5|D6|D7|D8|D9|D10|D11|D12|D13|
D14|D15|D16|D17|D18|D19|D20|D21|D22|D23|D24|
D25|D26|D27|D28|D29|D30|D31|D32|D33|D34|D35|
D36|D37|D38|D39|D40|D41|D42|D43|D44|D45|D46|
D47|D48|D49|D50|D51|D52|D53|D54|D55|D56|D57|
D58|D59|D60|D61|D62|D63|C0|C1|C2|C3|ICR|ACR|
ALR|AMR|AIN|MC0|MC1|MC2|TRI|TR0

Semantics:

A <CU register designator> denotes an addressable register in the CU. Each CU register designator symbolizes the 8-bit encoding of the address of a

register in CU memory. If the quadrant specifier is present, the leading two bits of the 8-bit field are assigned the specified number.

D0, D1, . . ., D63 Denote the 64 ADB locations.

C0, C1, C2, C3 Denote the 4 ACAR registers.

The remaining register mnemonics denote the register which they abbreviate. No spaces may appear within a CU register designator.

Examples:

$C0

$D32

$2D32

$ICR

3.2.6 Register Designators and Operand Fields for PE Instructions

PE OPERAND FIELDS

Syntax:

<blank PE operand> ::= <empty>

<PE address operand> ::= <ADR use indicator>
<address field> <ACARX> |
<address field> <ACARX> <ADR use> |
<register designator> <ACARX>

<literal PE operand> ::= <ADR use indicator>
<address field> <ACARX> |
<address field> <ACARX> <ADR use>

<routing operand> ::= <routing specifications> <ACARX>

<address field> ::= <arithmetic expression>

<ADR use indicator> ::= *|#|=|#*|*#

<ADR use> ::= ,<arithmetic expression> | <empty>

<routing specifications> ::= <arithmetic expression> |
<arithmetic expression> <routing distance>|
<PE register designator> |
<PE register designator> <routing distance>

<routing distance> ::= ,<arithmetic expression>

Semantics:

The <address operand> specifies the ACARX, ADR use and address field for those PE instructions which specify an operand address.

The <literal operand> specifies the ACARX, ADR use and address field for those PE instructions which do not require an operand but, rather, a shift count or bit number encoded in the address field of the instruction.

The <routing operand> is used in conjunction with only two instructions, RTG and RTL.

The <address field> sets the 16-bit address field of the instruction to the value of the arithmetic expression. The expression is evaluated using row arithmetic and may be either relocatable or absolute.

The <ADR use indicator> sets the ADR use field of the instruction. The convention used is as follows:

		ADR USE FIELD			
Symbol	Bits	13	14	15	Meaning
*		0	1	1	RGX indexing
#		1	0	1	RGS indexing
#|#		1	1	1	Combined indexing
=		0	0	0	Literal

The <ADR use> sets the ADR use field of the instruction to the value of the arithmetic expression. Word arithmetic is used in evaluating the

expression and the expression must be absolute. If the <ADR use> is <empty>, the ADR use field of the instruction is set to 1 (memory fetch--no indexing). Thus the ADR use field of the instruction may be set by either the <ADR use indicator> or <ADR use>.

A <register designator> causes one of two things to happen. If the specified register is a PE register, the ADR use field is set to 4 (register code) and the address field is encoded so as to specify the indicated register. If the specified register is an ACAR, the ADR use field is set to 0 (literal), the address field is set to 0, the ACARX field is set to the indicated ACAR and the enable bit set.

The <routing specifications> indicates the register connectivity and routing distance for the route instructions:

a) If a single <arithmetic expression> is used, ASK assembles a route of that distance, setting the register connectivity to the R register.

b) If the construct <arithmetic expression> <routing distance> is used, the first expression sets the register connectivity portion of the address field and the second sets the routing distance portion of the address field.

c) If only a <PE register designator> is used ASK sets the register connectivity portion of the address field to the indicated register and sets the routing distance portion of the address field to zero.

d) The construct <PE register designator> <routing distance> is self-explanatory.

The <ACARX> sets the ACARX field enable bit of the instruction to one and encodes the ACAR indicated by the value of the expression (taken modulo 4). Word arithmetic is used to evaluate the expression and the expression must be absolute.

Examples:

Address operand:

```
* X-1 (2)
# P2 (ACAR)
* MATRIX + (Q - R)
STUFF (2),3
MEMORY,1
MEMORY
= X + 14:8
= 0 (3)
$C3
$B
$R (2)
```

Literal operand:

```
SHIFTCOUNT
BITNUMBER (2),5
#BITNUMBER (2)
*(SHIFTCOUNT) (2)
```

Routing operand:

```
DISTANCE
2*WHICHREGISTER,DISTANCE
$S,DISTANCE
DIST (2)
CHUZREG,DIST (1)
$A,0 (2)
$A (2)
```

Address field:

PQ

PDQ + 2*N

3

-1

ADR use:

,3

,WHICHONE/2

,LITERAL + MAYBENOT

Routing specifications:

HERETOTHERE

REGISTER,24

$B,1

$A

Routing distance:

,DIST

,-1

,0

,NUMBEROFPES -1

REGISTER DESIGNATORS IN PE

Syntax:

<register designator> ::= $<register mnemonic>

<register mnemonic> ::= A|B|D|R|S|X|C0|C1|C2|C3

<PE register designator> ::= $<PE register mnemonic>

<PE register mnemonic> ::= A|B|R|S|D|X

Semantics:

A <PE register designator> denotes a register in the PE.

In addition, a <register designator> can denote the common data bus as defined by the contents of a specified ACAR. A <PE register designator> causes the encoding for that register to be placed in the address of the instruction. If a <register designator> specifies an ACAR, the address field of the instruction is set to zero, the ADR use field is set to zero (literal) and the ACARX field is set to the specified ACAR and the ACARX field enable bit is set.

Examples:

$A

$X

$C1

3.2.7 Operand Fields for Mode-Setting Instructions

Each PE has as one of its functional registers a so-called "Mode Register". Each Mode Register is a configuration of eight flip-flops designated mnemonically: E, E1, F, F1, G, H, I, J. The E and E1 bits are the enable bits for the PE (seen here as two separable 32-bit arithmetic units, one enabled by E, the other by E1). They serve effectively as on-off switches for the PEs.

The F and F1 bits associate with the PE arithmetic unit in similar fashion to the E and E1 bits and serve as the arithmetic fault bits (exponent overflow, etc.).

The G, H, I, J bits serve (in the pairs I-G, J-H) as utility bits and are set as a consequence of certain compare operators, or from a designated bit from the "A" Register, etc.

The syntax and semantics for the operand fields of instructions which manipulate these bits follows:

Syntax:

<mode pattern operand> ::= <arithmetic expression> <ACARX> |
<ACAR designator>

<mode setting operand> ::= <left mode specifier> <mode operator>
<right mode specifier> <ACARX>

<ACAR designator> ::= $C0|$C1|$C2|$C3

<left mode specifier> ::= <mode bit> | -<mode bit>

<mode bit> ::= E|E1|F|F1|G|H|I|J

<mode operator> ::= AND|OR|.AND.|.OR.

<right mode specifier> ::= E|E1|-E|-E1

Semantics:

The <mode pattern operand> is used in conjunction with the mode-bit loading mnemonics (LD-). In these instructions, the ILLIAC IV hardware ignores the ADR use field, i.e., the address field is treated as a literal and is ACAR indexable.

The <mode setting operand> is used in conjunction with the mode setting mnemonics (SET-). The address field of the instruction is encoded for the same operation as is indicated by the operand field. The convention -<mode bit> means the logical negation of the specified mode bit.

If the mode operators AND or OR are used a space must immediately precede and succeed them.

Examples:

Mode pattern operand:

1

0

0 (2)

$C2

Mode setting operand:

E OR E1

I AND -E (2)

H .OR. -E1

3.2.8 ASK Pseudo Operations

A pseudo operation constitutes an instruction to ASK which may or may not generate ILLIAC IV code. The general syntax for <ASK pseudo-op> is given below:

<ASK pseudo-op> ::= <EQU pseudo> |
<SYL pseudo> |
<WDS pseudo> |
<BLK pseudo> |
<FILL pseudo> |
<SET PE pseudo> |
<DATA pseudo> |
<ORG pseudo> |
<CHWS pseudo> |
GLOBAL |
LOCAL |
<DEFINE pseudo>*

3.2.8.1 EQU Pseudo

Syntax:

<EQU pseudo> ::= <label list> EQU <arithmetic expression> |
<register label list> EQU <register> |
<CU label list> EQU <CU register designator>

---

*See also section 4.5

<register label list> ::= <register symbol>: |

<register label list> <register symbol> :

<register> ::= <CU register designator> | <PE register designator>

<CU label list> ::= <register symbol>: | <CU symbol> : |

<CU label list> <register symbol>: |

<CU label list> <CU symbol>:

Semantics:

The function of EQU is to assign a value to the symbol(s) which label it. If an arithmetic expression is used, the value of the expression (evaluated using word arithmetic) is put into the word field portion of the symbol's value. The syllable bit is set to zero. If a <register> is used, the register symbol(s) are made to denote the same register as the one specified by <register>; additionally, if the <register> is a <CU register> then CU symbols may be assigned its address in CU-Memory.

Restrictions:

All symbols in the label list must not have been previously defined.

All symbols in the operand field must have been previously defined.

3.2.8.2 SYL Pseudo

Syntax:

<SYL pseudo> ::= <optional label list> SYL <SYL operand>

<optional label list> ::= <label list> | <empty>

<SYL operand> ::= <arithmetic expression> | <empty>

Semantics:

The SYL pseudo operation serves to reserve a block of 32-bit syllables. A label list is optional. If any labels are present, they receive the value of

the allocation counter at the time the SYL pseudo is encountered. ASK then emits the number of no-ops indicated by the value of the arithmetic expression (evaluated using word arithmetic), i.e., the requested block of 32-bit syllables is filled with no-ops. The value of the arithmetic expression must be absolute. If the <SYL operand> is <empty>, an expression value of zero is assumed.

Examples:

X: SYL 31

CURRENTACVALUE: SYL

3.2.8.3 WDS Pseudo

Syntax:

<WDS pseudo> ::= <optional label list> WDS <WDS operand>

<WDS operand> ::= <arithmetic expression> | <empty>

Semantics:

The WDS pseudo operation serves to reserve a block of 64-bit words, of length equal to the value of the arithmetic expression (evaluated using word arithmetic). The allocation counter is first adjusted to a 64-bit word boundary (even syllable), if necessary. If an adjustment is made, a no-op is placed in the syllable which is skipped over. At this point, all labels receive the value of the allocation counter (the label list is optional). The block of 64-bit words is then created by filling the appropriate number of words with zeros. The allocation counter then points to the next available 32-bit syllable at the end of the block of 64-bit words. The value of the arithmetic expression must be absolute. If the <WDS operand> is <empty>, the expression value of zero is assumed.

Examples:

```
P:  WDS
Q:  WDS 64
    WDS
```

3.2.8.4 BLK Pseudo

Syntax:

<BLK Pseudo> ::= <optional label list> BLK <BLK operand>

<BLK operand> ::= <arithmetic expression> | <empty>

Semantics:

The BLK pseudo operation serves to reserve a block of 4096-bit "words", i.e., rows of 64-bit words across PE memory. The number of rows is determined by the value of the arithmetic expression (evaluated using word arithmetic). If necessary, ASK adjusts the allocation counter to a quadrant boundary, filling in no-ops if the adjustment has to take place. All labels then receive the value of the allocation counter. The requested number of "words" is then spaced over (inserting zeros) and the allocation counter is set to the next available syllable beyond the requested block of storage. The allocation counter will point to a quadrant boundary after "execution" of this pseudo.

If the <BLK operand> is <empty>, the expression value of zero is assumed.

Examples:

```
X:  BLK 64
    BLK
```

3.2.8.5 FILL Pseudo

Syntax:

<FILL pseudo> ::= <optional label list> FILL <FILL operand>

<FILL operand> ::= <arithmetic expression> | <empty>

Semantics:

Let V be the value of the arithmetic expression. V determines a nonzero power of two, M, which is the smallest power of two not less than V. The directive to the assembler is to adjust the allocation counter to a position--syllable address--such that the allocation counter is congruent to V modulo M. Word arithmetic is used in evaluating the arithmetic expression. If the value of the expression is zero or if the operand field is empty, M is defined as being equal to 2. If the allocation counter has to move, no-ops are filled into the syllables skipped over. Labels are optional and, if any are present, receive as their value the value of the allocation counter after adjustment.

Examples:

```
        FILL 2          Even syllable
        FILL 7          Seventh syllable in a block of 8
      X: FILL 16        Head of a block of 16 syllables
```

3.2.8.6 SET Pseudo

Syntax:

<SET pseudo> ::= <label list> SET <arithmetic expression> |
<label list> SET |
<register label list> SET <register> |
<CU label list> SET <CU register designator>

Semantics:*

The SET pseudo performs analagously to the EQU pseudo, with the following differences:

a) No multidefinedness check is made on the symbol(s) being defined, i.e., one or more symbol(s) in the "label field" may have previously been defined.

b) The label(s) is redefined at the same point in the program in Pass II.

c) If the operand field is empty, the symbol(s) is defined with the current value of the allocation counter.

3.2.8.7 DATA Pseudo

Syntax:

<DATA pseudo> ::= <optional label list> DATA <data operand>

<data operand> ::= <data list>

<data list> ::= <data list element> |

<data list>, <data list element>

<data list element> ::= <number> | <symbol> | <string> |

(<data list>) <repeat part>

<repeat part> ::= <arithmetic expression>

Semantics:

The DATA pseudo operation provides for the loading of data into PE memory. A label list is optional. If necessary, the allocation counter is first adjusted to a word boundary and a no-op is inserted in the skipped syllable. The specified data is then placed in PE memory as 64-bit words.

---

*Semantics are given for only the first two forms, as the last two have not been implemented yet.

If a number is used, it's converted value (64-bit) is placed in memory

If a symbol is used, the value of its syllable field is placed in memory, right justified, in a field of zeros.

A repetitive list is placed in memory element by element, repeated as many times as is indicated by the value of the repeat part (word arithmetic).

Examples:

```
        DATA    -1
STUFF:  DATA    2, 3, 1.2, 01.3 @-8, (1, -1) N-1, X, 774:8
```

3.2.8.8 ORG Pseudo

Syntax:

<ORG pseudo> ::= <optional label list> ORG <arithmetic expression>

Semantics:

The ORG pseudo operation sets the allocation counter to the value of the arithmetic expression. Any labels are also given this value (in the syllable field). The expression is evaluated using syllable arithmetic. The allocation counter will have the same relocatability as the value of the expressi i.e., symbols defined by labeling an ILLIAC IV instruction will henceforth be absolute or relocatable, depending upon whether the value of this expression is absolute or relocatable.

Examples:

```
ORG @@ + 3
ORG X
```

3.2.8.9 CHWS Pseudo

Syntax

<CHWS pseudo> ::= <optional label list> CHWS <arithmetic expression>

Semantics:

The CHWS pseudo operation emits one ILLIAC IV instruction which sets the word size bit in the ACR register for 32 or 64 bit arithmetic in the PE's. The setting of this bit is according to the value of the arithmetic expression (word arithmetic).

Value of Expression	Word Size Setting Generated
0	64 bit
1	32 bit
32	32 bit
64	64 bit
Anything Else	Undefined

Examples:

CHWS 64

CHWS 1

3.2.8.10 LOCAL Pseudo

Semantics:

This pseudo-operation causes ASK to assemble CU instructions in the local mode unless

1) A GLOBAL pseudo-operation appears later, or
2) A CU instruction has a non-empty <global-local specifier>, in which case that instruction only is assembled with the indicated global-localness.

3.2.8.11 GLOBAL Pseudo

Semantics:

This pseudo-operation causes ASK to assemble CU instructions in the Global mode unless

1) A LOCAL pseudo-operation appears later, or
2) A CU instruction has a non-empty <Global-local specifier>, in which case that instruction only is assembled with the indicated Global-localness.

3.2.8.12 DEFINE Pseudo

Syntax:

<define pseudo> ::= DEFINE <define part>

<define part> ::= <define element> |
<define part>, <define element>

<define element> ::= <define identifier> =
<define text> ##

<define identifier> ::= <identifier>

<define text> ::= {any sequence of characters not including the character ## unless enclosed in string quotes}

Semantics:

The define pseudo causes the <define identifier> to serve as an abbreviation for the text bracketed by the = and the ##. From that point on in the program, whenever the <define identifier> is written, ASK will substitute for it the <define text> with which it is associated.

Restrictions:

1) The <define text> must not contain any unmatched " symbols.

2) A define identifier may not appear as a PE or CU register mnemonic.

3) A define identifier may be used alone as a <mode operand> but may not be used alone as a <left mode specifier> <mode operator> or <right mode specifier>.

Example:

```
DEFINE
LASTWORD = FILL 126;  WDS ##,
       Y = 3 ##;
       X:  LASTWORD Y;
```

is the same as:

```
       X: FILL 126;  WDS 3;
```

4. EXTENSIONS TO ASK - THE MACRO ASSEMBLER

Chapters 1-3 have described ASK as it exists at the time of this writing. This chapter describes the features of ASK, soon to be implemented, which will transform ASK into a powerful Macro Assembly System.

4.1 Definitions of the Tasks of Each Pass

4.1.1 Pass I

The task of Pass I of ASK is most concisely defined as: to determine the size of the ASK program and to define all symbols given by the user. Any pass of the assembler which performs those two functions will be designated as a Pass I (a non-trivial point since there may be several partial Passes I).

Implications of the above definition of Pass I enable us to give a more detailed accounting of the events of Pass I than was heretofore possible.

One implication of this definition is that ASK performs all pseudo operations which affect either of the two stated functions. This actually includes every pseudo operation mentioned in Chapter 3 (save GLOBAL and LOCAL), since each of them either defines some symbol or changes the value of the allocation counter in a way which must be known to Pass I, or both. ILLIAC IV instructions need not be processed in Pass I, except for defining their labels, since they affect the allocation counter in a known way, i.e., it is known from the mnemonic how many 32-bit instruction syllables a particular instruction will occupy. The performing of a pseudo operation, however, entails evaluating its operand field, which, therefore, must be evaluable in Pass I. Hence the statement: All operand fields of all pseudo operations must be Pass I evaluable.

A second implication of the above definition is that all input determining control statements must be performed in Pass I.

A third implication is that all defines must be expanded in Pass I, the text being a part of the input to ASK.

If there are one or more partial Passes I, they will not cause the input string to be scanned again. Additionally, the input string is not scanned in Pass II.

4.1.2 Pass II

The task of Pass II of ASK is to produce an ILLIAC IV relocatable code file. This implies that all operand fields of all ILLIAC IV instructions must be evaluated and placed in the proper field of the instruction syllable itself.

A secondary function of Pass II is to produce a listing for the user. The listing includes the source card image, an ASK-supplied sequence number, and the value of the allocation counter paired with the instruction which occupies that position. The presence of the instruction requires that the listing be produced in Pass II; were the instruction to be removed from the listing, the listing could be produced in Pass I. Since assemblers have by tradition produced listings in Pass II, including the instruction generated, it might be useful to examine that position more closely rather than to simply accede to the demands of tradition:

(1) It is of no concern to the user what instructions are generated by the assembler.

If the system under which the object program is running can provide an instruction counter setting, a memory dump, and a map of the contents of memory at the time of program termination, having the instruction allows one only to verify that the instruction was really there. For assembly level

programs, this is actually a useful alternative to have, a frequent cause of termination of assembly language programs being that of overwriting program with data and then executing it.

(2) It is useful and sometimes necessary when analyzing dumps to know what appearance code segments should have. This statement appeals to the reasons for rejecting statement (1).

At this point, consideration of the ILLIAC IV system is in order. A complete memory dump in octal of ILLIAC IV would require approximately 440 pages of computer paper and take approximately 20 minutes of printer time to print (assuming a 1200 LPM printer), for a single quadrant of ILLIAC IV. The impracticality of such a resource-consuming entity as a memory dump will probably preclude its existence except as a memory-image which resides in secondary storage for analysis. If there exists an interactive dump analyzer which can display to the user any memory location in a variety of formats, then a sufficient instruction printout would be the instruction mnemonic and its location -- which could be furnished in Pass I.

(3) An actual instruction printout is necessary in order to verify that the instructions are assembled correctly.

This is actually the reason for Pass II to produce the listing, which includes the assembled instructions. ASK will probably be in various stages of debugging for some time, as any modification to it reinitiates a short debugging period. Thus, the instruction listing being helpful in ensuring confidence in the performance of the assembler, the listing must be generated in Pass II.

4.1.2.1 Implementation of Pass II -- K-Machine

Pass II of ASK will be implemented by means of a simulated machine (the K-Machine), a program for which is constructed in Pass I. Figure 3

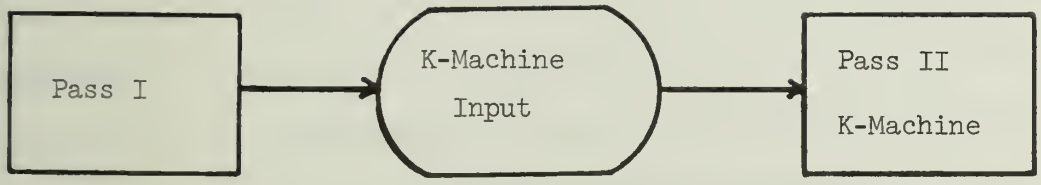


Figure 3. Relationship Between Pass I and Pass II.

depicts the relationship of the two passes. A complete description of the K-Machine is given in Appendix B; a less detailed description is given here.

The K-Machine maintains an automatic stack mechanism for the storage of operands. The stack mechanism facilitates the evaluation of arithmetic expressions, their K-Machine code equivalents being isomorphic to their polish postfix form. The arithmetic operators make use of the available bits in a B5500 word in excess of the 24 bits required for an ILLIAC IV address to carry the relocatability, externalness and arithmetic mode of the operands in the stack.

Several auxilliary registers exist, two of which are the Loader Information Register and the Instruction Register. Special K-Machine operators store the top of stack into all or portions of these registers. During the execution of a store into a field of the Instruction Register, the bits of the word in the top of stack which describe its value are examined and checked for validity. Certain fields of the instruction must be fixed at assembly time, that is, be Absolute; others, such as address fields, may contain instances of the most general values provided for within the assembler/loader system, i.e., Relocatable or External.

An instruction in the K-Machine causes one syllable of code to be emitted to the object code file from the Instruction Register. The instruction syllable and the loader information for that instruction are joined together before being placed in the code buffer.

The K-Machine fetches its instructions directly from a disk file and fetches its operands from the assembler's symbol table. Operands may be stored into the symbol table as well, allowing symbols to change in value during the second pass of the assembly, via the SET pseudo operation (section 3.2.8.6) or the assembly-time assignment statements (section 4.2).

The K-Machine is sufficiently powerful that it can perform many Pass I operations as well as Pass II, given a properly constructed symbol table and the knowledge that it is performing a Pass I rather than Pass II. The implementation of Conditional Assembly uses that property and is discussed in section 4.6.

4.2 Assembly-Time Assignment Statements

The SET pseudo resembles an assignment statement closely enough that the user may as well have that facility directly. Additionally, the arithmetic assignment statement may then be embedded in the construct <primary>. The following changes to the present syntax and semantics will be implemented in ASK:

Syntax:

```
<statement> ::= . . . | <register assignment statement> |
                        <arithmetic assignment statement>
<register assignment statement> ::= <register symbol> := <register
                                    assignment> |
                                    <CU symbol> := <CU register assignment>
<register assignment> ::= <register> | <register symbol> :=
                          <register assignment>
<CU register assignment> ::= <CU register designator> |
                             <register symbol> := <CU register assignment> |
                             <CU symbol> := <CU register assignment>
<arithmetic assignment statement> ::= <PE symbol> := <arithmetic
                                      assignment > |
                                      <CU symbol> := <arithmetic
                                      assignment>
<arithmetic assignment> ::= <arithmetic expression> |
                            <arithmetic assignment statement>
```

<primary> ::= . . . | <arithmetic assignment statement>

Semantics:

The semantics of the assignment statements are the same as those of the SET pseudo operation. The arithmetic assignment statement is performed in both passes if it is a <statement> and in the same pass as the expression which contains it if it is a <primary>.

4.3 Allocation Counters

ASK will maintain sixty-four allocation counters for use during an assembly. The allocation counters are numbered 0-63 and code assembled under separate allocation counters appears in that order in the object code file, i.e code assembled under allocation counter 0 followed by code assembled under allocation counter 1 etc. Each allocation counter is initially set to Relocatabl zero and is advanced during the normal course of assembling instructions, etc. under the control of that particular allocation counter (abbreviated AC). The syntax for assembling a section of code under a particular AC is given here It is an enlargement of the syntax for <statement> given in section 3.2.4:

<statement> ::= . . . | <compound statement> | <block>

<block> ::= BEGIN BLOCK <allocation counter part> <compound tail>

<allocation counter part> ::= <empty> | USE <allocation counter>

<allocation counter> ::= <arithmetic expression> | *

<compound tail> ::= <statement> END | <statement> ; <compound tail>

<compound statement> ::= BEGIN <allocation counter part> <compound tail>

Semantics:

For the moment we will ignore the distinction between <block> and <compound statement>. The allocation counter that the user desires to use is denoted by <allocation counter>. If <allocation counter> is an arithmetic

expression then the value of the expression determines the allocation counter to use, otherwise the AC numerically next will be used (*). The following three features of these constructs should be noted:

(1) Allocation counters may be switched in nested fashion.

```
Example:    BEGIN  USE  3
            . . . . .     % CODE ASSEMBLED UNDER AC 3
                      BEGIN  USE  2
                      . . . . . % CODE ASSEMBLED UNDER AC 2
                      END  ;
            . . . . . % CODE ASSEMBLED UNDER AC 3
            END
```

(2) Allocation counters may be reentered. Example:

```
BEGIN  USE  15
. . . . .
END  ;
. . . . .
BEGIN  USE  15
. . . . .
END
```

(3) Except as noted in (1) and (2), the scope of an AC is the <block> or <compound statement> whose <allocation counter part> designates its use, i.e., from BEGIN to matching END.

Multiple allocation counters can cause difficulties both for the assembler and for the user. The assembler's difficulties lie in the necessary overhead involved in keeping track of the ACs, such as sweeping the symbol table adjusting addresses. The user's difficulties lie in the rules he must adhere to in order to make use of multiple allocation counters. These

difficulties arise as a result of the following fact: In Pass I, except for allocation counter 0, ASK has no way of knowing the actual relocatable addresses assigned to a symbol defined under some allocation counter. This arises as a result of the fact that the actual origin of AC_i is known only in terms of the largest values of AC_j, $0 \leq j < i$, and the largest value of AC_0 is not known until the end of Pass I. The user must, then, observe the following restriction For any expression which must be evaluated in Pass I, if it contains relocatable symbols, they must be uniform as to the allocation counter under which they are defined. However, in expressions which are evaluated only in Pass II, relocatable symbols defined under different ACs may be included.

A sample program illustrates this restriction:

```
BEGIN      % USE 0 IS IMPLICIT
S:  JUMP  T ;
        BEGIN USE 2
        A: BLK 2 ;
        B: BLK 21 ;
        C: BLK B-A ; % LEGAL
        E: EQU A ; % LEGAL
        R: EQU S ; % LEGAL
        P: EQU A-S ; % ILLEGAL, BUT...
        T: SLIT(0) = A-- S ; % LEGAL SINCE PASS II EVALUATION
        JUMP  V ;
        END ;
V:
END S.
```

4.4 Lexicographical Level at Assembly-Time

ASK will provide a facility whereby the user has at his disposal levels of nomenclature at assembly time. This facility, among other possibilities, allows macros, if the user so desires, to be completely self-contained with no problems of conflicting with symbols already defined elsewhere by the user. Upon the occurrence of the "BEGIN BLOCK" construct, ASK reduces the level of nomenclature by one and upon encountering the matching END, increases it by one. At any level of nomenclature, i.e., block, reference may be made to either symbols at a higher level or symbols local to that block.

ASK has at all times a set of symbols visible to it. A symbol is said to be defined with respect to a reference to it as an operand if it is both visible and possesses a value. Circumstances may arise in which the set of symbols visible to ASK and the set of symbols the user desires to be visible do not agree. One such circumstance involves forward references to symbols defined local to a block. If the symbol is already defined at the time of the reference, ASK considers the reference as one to a symbol at a (possibly) higher level than the present one. Hence, its actions may not correspond to the desires of the user who would like ASK to "see" the local symbol. In order to resolve this difficulty, ASK provides a pseudo declaration which enables the user to make local symbols (i.e., which must be forward referenced) visible to ASK without actually defining them. The syntax for this pseudo declaration is:

```
    LOCAL <identifier list>
<identifier list> ::= <identifier> | <identifier list> , <identifier>
```

The following example may serve to illustrate this point:

```
BEGIN BLOCK
    X: EQU 15 ;
    BEGIN BLOCK
```

```
        SLIT(0) = X ;
        LOCAL X ;
        SLIT(1) = X ;
        X:: BLK 1
    END
END
```

In the example, the two SLIT instructions refer to two different X's.

4.5 Defines, Pseudo-Strachey Macros

The macro facility of ASK deviates from the traditional usage of the term in which one spoke of "macro instructions". In this sense, one was limited to defining "macros" which in their usage appeared as machine "instructions" with some sort of operand field. In ASK the notion of Strachey's [4] that macros can be seen as parameterized abbreviations for text is implemented. Thus, for a "macro" in ASK, the definition is some text which is substituted for the "macro" identifier upon its occurrence, and the term "define" is used instead of "macro".

A DEFINE definition in ASK has the following appearance:

DEFINE <DEFINE identifier> <optional parameter list> = <DEFINE text> ##

The parameter list is a list of identifiers enclosed in parentheses. The defin text is an arbitrary string of symbols, with one exception: If the word DEFINE occurs in the define text, it must have a corresponding "##" also in the define text; and likewise within the inner DEFINE - ## pair. This rule is not really restriction since it allows the possibility of a define, upon its expansion, to generate another define -- and then expand it if the user so desires. Defines may be called in nested fashion to a depth of 32.

At the point of invocation of a define, a number of dummy defines are constructed -- one corresponding to each parameter of the define being

expanded. ASK then scans the definition text as input, linking into parameters as if they were defines, until the text is terminated. It then resumes the assembly from the point in the original input after the appearance of the define identifier and its actual parameters. The actual parameters of a define may be any text with observation of the following rule: If a comma (,) appears in the text it will be construed as a parameter delimiter unless it is enclosed in parentheses or square brackets. (Note: in the B6500 implementation in which the EDCDIC character set will be available, any text enclosed between pairs of vertical bars will be considered as part of the text of a parameter, but the vertical bars will not themselves be passed as parameter text, thus allowing "free" commas to be passed to a define.)

ASK uses secondary storage (disk) as backup for define text, so that the limit to the length of a define is determined only by the availability of disk storage. ASK also provides for the contingency that the text of a define may originate on the disk. Hence two possibilities arise:

(1) System defines. ASK's symbol table can be initialized with various previously declared defines. These defines can interface the user to OS4 and can be modified without recompiling ASK.

(2) User define libraries. The user could keep a file on disk of definitions, define identifiers and formal parameters, with a suitable directory, that he could refer ASK to at any point in an assembly. If a system program is written to properly maintain these libraries, inter user communication and dissemnation of useful routines could be facilitated.

4.6 Conditional Assembly

ASK will provide for the possibility of assembling sections of program conditionally dependent upon relationships which exist between symbols

whose values are known at assembly time. The general implementation plan for conditional assembly is to use the K-Machine, which was designed to perform Pass II of ASK, during Pass I whenever the input to Pass II is conditioned by the user. The constructs which constitute the conditional are compiled into K-Machine language, and the K-Machine is called in to execute the compiled code during Pass I. The code is so constructed that the effect of the K-Machine is to add additional K-Machine instructions to the Pass II input file (See Figure 4).

Our attention should center on two control flip/flops in the K-Machine which are important in the controlling of its functions. One is the EFF, the Execute Flip/Flop, which partially controls the execution of instructions in the K-Machine; the other is the CFF, the Copy Flip/Flop, which partially controls the execution of instructions and governs completely the copying of instructions over to the Pass II input file. These two flip/flops and the designated Pass (I or II) form two Boolean expressions, one enabling the execution of instructions, the other the copying of instructions to the Pass II input file. The two Boolean expressions are:

(1) PASSII OR (CFF IMP EFF)

(2) PASSI AND CFF

Note that in Pass II, (1) is always TRUE and (2) is always FALSE. In Pass I the expressions reduce to:

(1) CFF IMP EFF

(2) CFF

Due to the nature of implication, instructions will be executed if EFF=TRUE. If EFF=FALSE, then the following statement holds: Instructions copied to the Pass II input file (CFF=TRUE) are not executed and instructions not copied to the Pass II input file (CFF=FALSE) are executed.

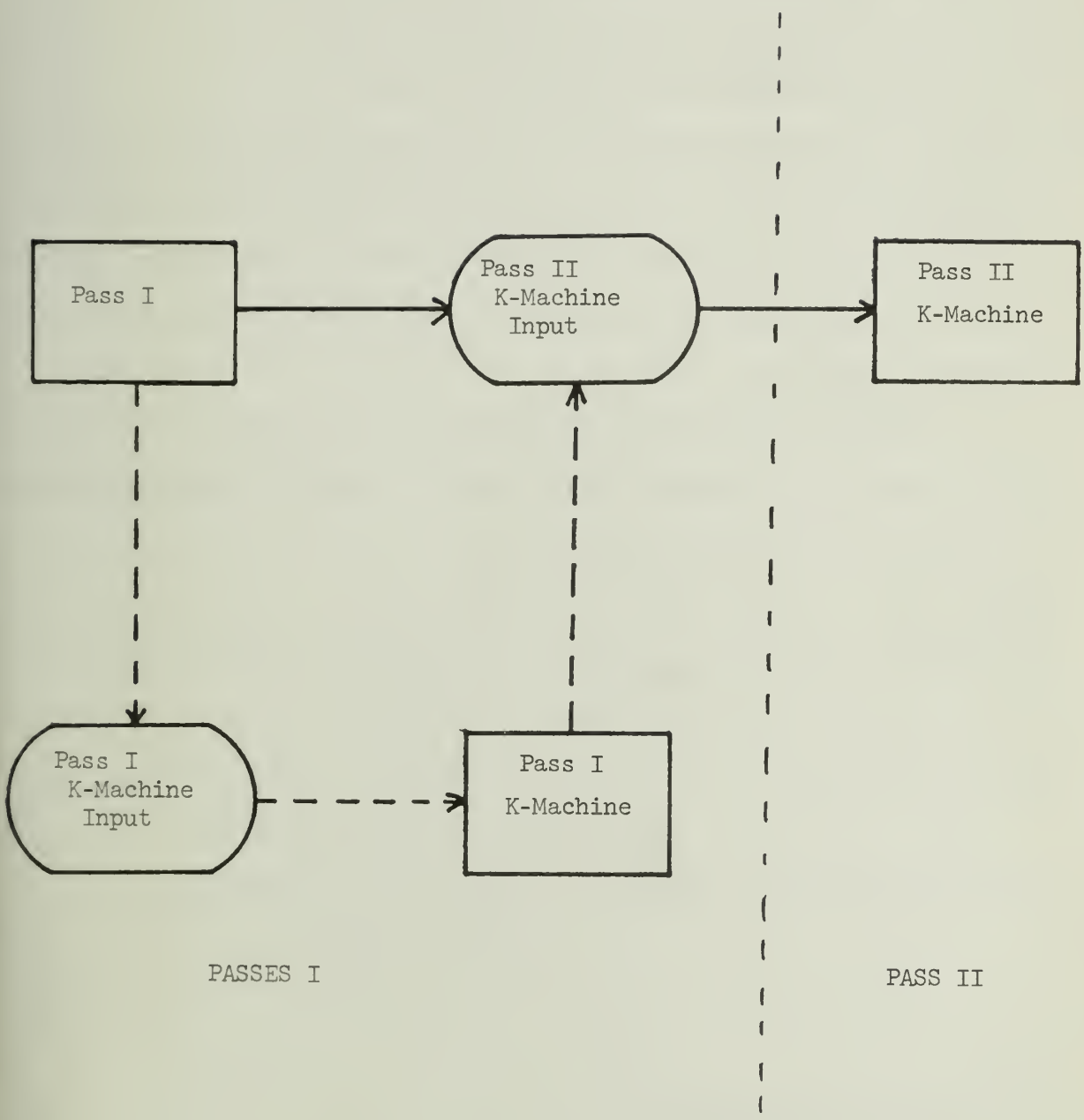


Figure 4. Relationship of Pass I and Pass II for Conditional Constructs

From the above, it can be seen that K-Machine instructions fall into three classes:

(1) Instructions executed in Pass I only

(2) Instructions executed in Pass II only

(3) Instructions executed in both passes

Which class a particular instruction is in depends totally upon context. More interesting than the contexts which determine the classes for K-Machine instructions are the constructs which generate instructions of one or more of these classes. We will concern ourselves with the instructions generated for various conditional constructs.

The syntax for conditional constructs is given here. The individual syntax, semantics, and implementational aspects of these constructs are treated in the sections which immediately follow.

Syntax:

<conditional construct> ::= <conditional expression> | <conditional statement construct>

<conditional expression> ::= <conditional arithmetic expression> | <conditional Boolean expression>

<conditional statement construct> ::= <conditional statement> | <WHILE statement> | <DO statement>

4.6.1. Conditional Statements

Syntax:

<statement> ::= . . . | <conditional statement>

<conditional statement> ::= <if clause> <statement> | <if clause> <statement> ELSE <statement>

<if clause> ::= IF <Boolean expression^{x}> THEN

Semantics:

A conditional statement indicates that the instructions to be assembled are to be dependent upon the logical result of evaluating a Boolean expression. If the Boolean expression is TRUE, the statement following the THEN is assembled, otherwise either the statement following the ELSE is assembled or, if the ELSE is not present, no instructions are assembled as a result of the conditional statement. When conditional statements are nested, the pairing of THENs and ELSEs can be determined by the following rule:

> For any THEN, the "matching" ELSE is the leftmost "unmatched" ELSE not separated from the THEN by any other "unmatched" THEN. If any "unmatched" THEN separates a THEN and an ELSE, the former THEN is also "unmatched".

Example:

```
IF   THEN   IF   THEN   ELSE   IF   THEN   ELSE
     (           (         )        (         )
```

Any Boolean expression which occurs in an <if clause> must be Pass I evaluable.

The K-Machine code for a conditional statement is generated into the input file for the Pass I K-Machine. The K-Machine is called into execution after the outermost conditional construct has been successfully compiled.

The general skeleton of the K-Machine code generated for a conditional statement is given in Figure 5 . The assumption is made that the code for the statement will be copied into Pass II. Should this assumption prove to be false, i.e., if the statement is or contains any conditional constructs or pseudos, then the copying of instructions will be enabled and disabled in accordance with the individual statement(s).

---

*<Boolean expression> is not defined in this document. The implementation is certain to include relations, logical operations and special constructs which will enable the user to interrogate the state of the assembly, i.e., is a symbol Absolute, is it word aligned, etc.

SOURCE LANGUAGE	SKELETAL K-MACHINE CODE	REMARKS
IF	CPYD	Disable CFF so the next Code is not copied into the Pass II input file
<Boolean Expression>	(Code for Boolean Expression) LITC (Branch Distance to *) BFC *	Branch forward to ELSE if B.E. is false
THEN	CPYE	Let the Code for the statement be copied into Pass II.
<statement>	(Code for the statement) CPYD LITC (Branch Distance to #) BF	Disable copying for the branch Branch around the ELSE
ELSE	* CPYE	
<statement>	(Code for the statement) # CPYE	

Figure 5. Skeletal K-Machine Code for Conditional Expressions.

4.6.2 Iterative Statements

ASK will provide a means whereby statements may be assembled iteratively. Two constructs are planned to be implemented, the WHILE statement and the DO statement.

4.6.2.1 WHILE - DO

Syntax:

<WHILE statement> ::= WHILE <Boolean expression> DO <statement>

Semantics:

The WHILE statement indicates that the statement following the DO is to be assembled if, and as long as, the Boolean expression remains TRUE. The semantics of the WHILE statement are easily made precise by way of a pseudo-Algol definition:

```
begin L:  if <Boolean expression> then
              begin <statement>; go to L end
end
```

The WHILE statement causes K-Machine code to be generated into the Pass I K-Machine input file for execution at the proper time. A skeletal diagram of the code generated is given by Figure 6.

SOURCE LANGUAGE	SKELETAL K-MACHINE CODE	REMARKS
WHILE	CPYD	The code for the Boolean expression is not input to Pass II
<Boolean Expression>	# (Code for Boolean Expression) LITC (Branch Distance to *) BFC *	
DO	CPYE	
<statement>	(Code for the statement) CPYD LITC (Branch distance to #) BB # * CPYE	The branch is not copied into Pass II

Figure 6. Skeletal K-Machine Code Generated for WHILE Statement.

4.6.2.2 DO - UNTIL

Syntax:

<DO statement> ::= DO <statement> UNTIL <Boolean expression>

Semantics:

The DO statement indicates that the statement following the DO is to be assembled once and then repeatedly until the Boolean expression becomes TRUE. A pseudo-Algol definition of these semantics is

begin L: <statement>; if not (<Boolean expression>) then go to L end

The DO statement is implemented through the use of the K-Machine, the skeletal code for which is given in Figure 7.

SOURCE LANGUAGE	SKELETAL K-MACHINE CODE	REMARKS
DO	* CPYE	The code for the statement is copied into Pass II. . .
<statement>	(Code for the statement)	
UNTIL	CPYD	The code for the Boolean expression is not copied into Pass II
<Boolean Expression>	(Code for the Boolean expression) LITC (Branch distance to *) BBC * CPYE	

Figure 7. Skeletal K-Machine Code Generated for DO Statement.

4.6.3 Conditional Expressions

Syntax:

<conditional expression> ::= <conditional arithmetic expression> |
<conditional Boolean expression>

<conditional arithmetic expression> ::= <if clause> <arithmetic expression>
ELSE <arithmetic expression>

<conditional Boolean expression> ::= <if clause> <Boolean expression>
ELSE <Boolean expression>

<primary> ::= . . . | <conditional arithmetic expression>

<Boolean primary> ::= . . . | <conditional Boolean expression>

Semantics:

The conditional expression indicates that the evaluation of an expression depends upon the logical value of a Boolean expression. The rules for determining which expression is evaluated and the proper pairing of THENs and ELSEs are similar enough to those given in Section 4.6.1, Conditional Statements, to be omitted here.

The evaluation of conditional expressions presents a unique problem for ASK. Consider the following three examples:

(1) Z: EQU 2+IF A LSS B THEN X ELSE Y ;

(2) Z: LDA 2+IF A LSS B THEN X ELSE Y ;

(3) Z: = 2+IF A LSS B THEN X ELSE Y ;

Example (1) must be evaluated in Pass I only; example (2) in Pass II only; and example (3) in both Pass I and Pass II. Further, the action of ASK in case (1) and (3) differs depending upon whether it occurs within a conditional construct, or is isolated as a single conditional construct itself. In the

former case, K-Machine code for the entire pseudo operation must be generated; in the latter case, K-Machine code must be generated for the expression only, the K-Machine invoked and the result returned. Case (1) is interesting in another aspect. The conditional expression occurs as the second term of the arithmetic expression. Since the expression is to be evaluated in Pass I only, ASK will attempt to evaluate it interpretively, saving an invocation of the K-Machine; hence the constant "2" will already be in the Pass I operand stack, the stack used for the interpretive evaluation of arithmetic expressions. The arithmetic expression parser uses the method of recursive descent, which allows for the following action to take place: The procedure which compiles the construct, <primary>, detects that a) there is a conditional expression, and b) it is in the interpretive mode. It then emits code to the K-Machine to duplicate the operand stack in the K-Machine stack, switches the mode of evaluation to generative, and compiles the conditional expression. The "+" is subsequently emitted as an operator, rather than actually being performed, the mode of evaluation having been changed.

Figure 8 shows the K-Machine code generated for conditional expressions. The code generated for case (3) above is the same as for case (2) except that it is preceded by an EXE (Execute Enable) operator and followed by an EXD (Execute Disable) operator. In this case, the only code copied into Pass II is the code to compute an unconditional arithmetic expression. The Boolean expression is not evaluated in Pass II.*

4.6.4 Listing Control

The task of producing a listing for the benefit of the user is complicated in the presence of the conditional assembly. It is desirable to

*In fact no Boolean expression is ever evaluated in Pass II.

SOURCE LANGUAGE	SKELETAL K-MACHINE CODE	REMARKS
IF	CPYD	None of this code is copied to Pass II
<Boolean Expression>	(Code for Boolean Expression)	
THEN	LITC (Branch Distance to *) BFC	
<Expression>	(Code for the Expression) LITC (Branch Distance to #) BF #	
ELSE		
<Expression>	*(Code for the Expression) # CPYE	

(a) Code Generated for Type 1) Contexts.

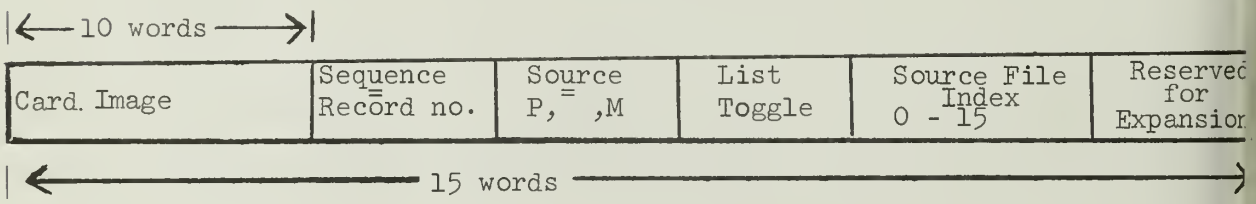
IF	CPYD	
<Boolean Expression	(Code for Boolean Expression) LITC (Branch Distance to *) BFC *	
THEN	CPYE	Allow code to be copied to Pass II
<Expression>	(Code for the Expression) CPYD LITC (Branch to #) BF *	Disallow copying for the branch
ELSE	* CPYE	
<Expression>	(Code for the Expression) #CPYE	

(b) Code Generated for Type 2) Contexts.
The code for type 3) is the same as in (b) but preceded by EXE and followed by EXD.

Figure 8. Code Generated for Conditional Expressions.

indicate to the user which sections of code have been assembled and which have not. At the same time, it is necessary that there be no confusion of the listing of the original source text and the listing which indicates that a certain branch has been taken. The listing control features of ASK will take into account the above factors together with indicating from which of several input files the source language originated, the possibility of inhibiting the listing altogether, and, in the case of a merge assembly, whether the card image was an update (patch) card of a card from the merge file.

ASK will build (in Pass I) a file containing the totality of the input to it. A record in this file will be formatted as follows:



The Card Image will be an exact copy of the image input to ASK. The Sequence number will be the ASK supplied sequential number of this card image; it corresponds to the record number of its record in the card image file. The source key will be used, in merge compiles, to indicate whether the card image originated in the merge file (M) or in the patch file (P). The source file index will correspond to the card input file and the 15 possible tape input files; this will enable the user to ascertain more precisely the origin of this card image. The List Toggle, if false, will inhibit the listing of this line altogether (unless an error occurred as a result of this card image, in which case it is listed unconditionally).

The K-Machine maintains a register (C) which gives the record number of the card image to be printed next. Pass I emits an instruction to Pass II

to read a specific record (n) from the card image file and prepare it for listing. The K-Machine takes one of the following actions depending upon the relationship that holds between n and the content of register C:

n less than C

This is the case when assembly language is being processed iteratively. The action is to read record n and if the Listing Toggle is true, print n as a sequence number, indicating that the card image is now being assembled. The content of register C is not changed.

n = C

Prepare for printing (conditionally upon the Listing Toggle) the card image currently in the "buffer".

n = C+1

Read record n; increment C and prepare the card image for printing (conditionally as above).

n greater than C+1

This is the case when card images have been "skipped over" due to a conditional construct which inhibited certain code from being assembled. The action taken is to read and print (conditionally) card images, incrementing C each time, until record n has been prepared for printing.

The above action will provide the user with enough information to determine which branches are being taken in his conditional assembly. Consider, as an example, that the user has coded a WHILE "loop". If the Boolean expression is true the first time it is executed, the listing will indicate the card images being assembled (by printing them) together with the ILLIAC IV instruction(s) generated by each card image. Thereafter, the card image sequence numbers only will appear in conjunction with the ILLIAC IV instruc-

tions generated, indicating that the loop is being repeated. Should the Boolean expression result in a value of FALSE the first time it is executed, however, the next instruction to print a line will be for a record which occurs later in the file (probably). This will result in a listing of the card images in between (the card images containing the WHILE loop) with no instructions generated indicating that those card images were not processed by ASK.

4.7 Errors - Termination of the Assembly

An error in any pass of ASK will cause the assembly to be terminated at the end of that pass. If the error occurs in Pass I, the listing will be generated up to the card image which contained the error if it is the first error encountered. The text of the error message will be printed immediately beneath the line on which the error occurred, together with the current content of the scan buffer, giving an indication of the location of the error on the card image. No K-Machine execution may subsequently be made since the K-Machine code will not in general produce well-defined results after the occurrence of a syntax error.

If an error is discovered by the K-Machine, the error message will be printed in a manner similar to those described above. However, any additional information will be dependent upon the K-Machine operator which detected the error. For example, if the operator is OPDC then the symbol table could be consulted for the text of the identifier whose appearance caused the OPDC to be generated.

An ASK program must be free from any errors for an ILLIAC IV code file to be generated.

5. SUMMARY

ASK represents the solution to the problem of assembling code for the ILLIAC IV. The problem is made more interesting by the fact that the assembler runs on a different machine (the Burroughs B6500) than the one for which code is being assembled, which allows the assembler to be written in a high level language (ALGOL) rather than bootstrapping itself onto the object computer.

The remote job entry facilities of the B5500/B6500 were primary considerations in making the ASK language a "free field" assembly language, as it is bothersome for users at remote terminals to build fixed field card image files.

The "free field"-ness of the language led to the decision that macros should be inline text substitutions in nature rather than simple macro instructions. Experience with Strachey's macro generator [4] and other Strachey-like macro generators reinforced this design decision.

An ever increasing facility and familiarity with ALGOL greatly influenced the specification of the conditional constructs in ASK and were responsible for the block structure features of the language.

In short, ASK is a natural and reasonable product of its environment, the Burroughs B6500 and the ILLIAC IV.

"Die welt ist alles, was der fall ist."

APPENDIX A

EXPANSION OF THE META-LINGUISTIC TERM <ILLIAC IV INSTRUCTION>

< ILLIAC IV instruction > ::=

AD	< PE address operand >	
ADA	< PE address operand >	
ADB	< PE address operand >	
ADD	< PE address operand >	
ADEX	< PE address operand >	
ADM	< PE address operand >	
ADMA	< PE address operand >	
ADN	< PE address operand >	
ADNA	< PE address operand >	
ADR	< PE address operand >	
ADRA	< PE address operand >	
ADRM	< PE address operand >	
ADRMA	< PE address operand >	
ADRN	< PE address operand >	
ADRNA	< PE address operand >	
ALIT	< ACAR selector >	<short literal operand > |
AND	< PE address operand >	
ANDN	< PE address operand >	
ASB	< blank PE operand >	
BIN	< ACAR selector >	< CU memory operand > |
BINX	< ACAR selector >	< CU memory operand > |
CAB	< literal PE operand >	
CACRB	< CU memory operand >	
CADD	< ACAR selector >	<CU memory operand > |

```
CAND        < ACAR selector >   < CU memory operand > |
CCB         < ACAR selector >   < CU memory operand > |
CEXOR       < ACAR selector >   < CU memory operand > |
CHSA        < blank PE operand > |
CLC         < ACAR selector >   < blank CU operand > |
CLRA        < blank PE operand > |
COMPA       < blank PE operand > |
COMPC       < ACAR selector >   < blank CU operand > |
COPY        < ACAR selector >   < CU memory operand > |
COR         < ACAR selector >   < CU memory operand > |
CRB         < ACAR selector >   < CU memory operand > |
CROTL       < ACAR selector >   < CU memory operand > |
CROTR       < ACAR selector >   < CU memory operand > |
CSB         < ACAR selector >   < CU memory operand > |
CSHL        < ACAR selector >   < CU memory operand > |
CSHR        < ACAR selector >   < CU memory operand > |
CSUB        < ACAR selector >   < CU memory operand > |
CTSBF       < ACAR selector >   < compare and skip operand > |
CTSBT       < ACAR selector >   < compare and skip operand > |
DUPI        < ACAR selector >   < CU memory operand > |
DUPO        < ACAR selector >   < CU memory operand > |
DV          < PE address operand > |
DVA         < PE address operand > |
DVM         < PE address operand > |
DVMA        < PE address operand > |
DVN         < PE address operand > |
DVNA        < PE address operand > |
DVR         < PE address operand > |
DVRA        < PE address operand > |
DVRM        < PE address operand > |
DVRMA       < PE address operand > |
DVRN        < PE address operand > |
DVRNA       < PE address operand > |
```

```
EAD       < PE address operand > |
EOR       < PE address operand > |
EQLXF     < ACAR selector >  < compare and skip operand > |
EQLXFA    < ACAR selector >  < compare and skip operand > |
EQLXT     < ACAR selector >  < compare and skip operand > |
EQLXTA    < ACAR selector >  < compare and skip operand > |
EQV       < PE address operand > |
ESB       < PE address operand > |
EXCHL     < ACAR selector >  < CU memory operand > |
EXEC      < ACAR selector >  < blank CU operand > |
FINQ      < blank CU operand > |
GB        < PE address operand > |
GRTRF     < ACAR selector >  < compare and skip operand > |
GRTRFA    < ACAR selector >  < compare and skip operand > |
GRTRT     < ACAR selector >  < compare and skip operand > |
GRTRTA    < ACAR selector >  < compare and skip operand > |
HALT      < blank CU operand > |
IAG       < PE address operand > |
IAL       < PE address operand > |
IB        < literal PE operand > |
ILE       < PE address operand > |
ILG       < PE address operand > |
ILL       < PE address operand > |
ILO       < blank PE operand > |
ILZ       < blank PE operand > |
IME       < PE address operand > |
IMG       < PE address operand > |
IML       < PE address operand > |
IMO       < blank PE operand > |
IMZ       < blank PE operand > |
INCRXC    < ACAR selector >  < blank CU operand > |
```

INR	< blank CU operand > |
ISE	< PE address operand > |
ISG	< PE address operand > |
ISL	< PE address operand > |
ISN	< blank PE operand > |
IXE	< PE address operand > |
IXG	< PE address operand > |
IXGI	< PE address operand > |
IXL	< PE address operand > |
IXLD	< PE address operand > |
JAG	< PE address operand > |
JAL	< PE address operand > |
JB	< literal PE operand > |
JLE	< PE address operand > |
JLG	< PE address operand > |
JLL	< PE address operand > |
JLO	< blank PE operand > |
JLZ	< blank PE operand > |
JME	< PE address operand > |
JMG	< PE address operand > |
JML	< PE address operand > |
JMO	< blank PE operand > |
JMZ	< blank PE operand > |
JSE	< PE address operand > |
JSG	< PE address operand > |
JSL	< PE address operand > |
JSN	< blank PE operand > |
JUMP	< short literal operand > |
JXE	< PE address operand > |
JXG	< PE address operand > |
JXGI	< PE address operand > |

```
JXL         < PE address operand > |
JXLD        < PE address operand > |
LB          < PE address operand > |
LDA         < PE address operand > |
LDB         < PE address operand > |
LDC         < ACAR selector >   < PE register specifier > |
LDD         < register designator > |
LDE         < mode pattern operand > |
LDE1        < mode pattern operand > |
LDEE1       < mode pattern operand > |
LDG         < PE address operand > |
LDH         < PE address operand > |
LDI         < PE address operand > |
LDJ         < PE address operand > |
LDL         < ACAR selector >   < CU memory operand > |
LDR         < PE address operand > |
LDS         < PE address operand > |
LDX         < PE address operand > |
LEADO       < ACAR selector >   < blank CU operand > |
LEADZ       < ACAR selector >   < blank CU operand > |
LESSF       < ACAR selector >   < compare and skip operand > |
LESSFA      < ACAR selector >   < compare and skip operand > |
LESST       < ACAR selector >   < compare and skip operand > |
LESSTA      < ACAR selector >   < compare and skip operand > |
LEX         < PE address operand > |
LIT         < ACAR selector >   < long literal operand > |
LIT         < ACAR selector > = < long literal operand > |
LOAD        < ACAR selector >   < CU memory operand > |
LOADX       < ACAR selector >   < CU memory operand > |
ML          < PE address operand > |
MLA         < PE address operand > |
MLM         < PE address operand > |
```

```
MLMA        < PE address operand > |
MLN         < PE address operand > |
MLNA        < PE address operand > |
MLR         < PE address operand > |
MLRA        < PE address operand > |
MLRM        < PE address operand > |
MLRMA       < PE address operand > |
MLRN        < PE address operand > |
MLRNA       < PE address operand > |
MULT        < PE address operand > |
NAND        < PE address operand > |
NANDN       < PE address operand > |
NEB         < PE address operand > |
NOR         < PE address operand > |
NORM        < blank PE operand > |
NORN        < PE address operand > |
OFB         < blank PE operand > |
ONESF       < ACAR selector >   < skip operand > |
ONESFA      < ACAR selector >   < skip operand > |
ONEST       < ACAR selector >   < skip operand > |
ONESTA      < ACAR selector >   < skip operand > |
ONEXF       < ACAR selector >   < skip operand > |
ONEXFA      < ACAR selector >   < skip operand > |
ONEXT       < ACAR selector >   < skip operand > |
ONEXTA      < ACAR selector >   < skip operand > |
OR          < PE address operand > |
ORAC        < ACAR selector >      < blank CU operand > |
ORN         < PE address operand > |
RAB         < literal PE operand > |
RTAL        < literal PE operand > |
RTAR        < literal PE operand > |
```

RTG < routing operand > |
RTL < routing operand > |
SAB < literal PE operand > |
SAN < blank PE operand > |
SAP < blank PE operand > |
SB < PE address operand > |
SBA < PE address operand > |
SBB < PE address operand > |
SBEX < PE address operand > |
SBM < PE address operand > |
SBMA < PE address operand > |
SBN < PE address operand > |
SBNA < PE address operand > |
SBR < PE address operand > |
SBRA < PE address operand > |
SBRM < PE address operand > |
SBRMA < PE address operand > |
SBRN < PE address operand > |
SBRNA < PE address operand > |
SETC < ACAR selector > < mode bit specifier > |
SETE < mode setting operand > |
SETE1 < mode setting operand > |
SETF < mode setting operand > |
SETF1 < mode setting operand > |
SETG < mode setting operand > |
SETH < mode setting operand > |
SETI < mode setting operand > |
SETJ < mode setting operand > |
SHABL < literal PE operand > |
SHABML < literal PE operand > |
SHABMR < literal PE operand > |

```
SHABR       < literal PE operand > |
SHAL        < literal PE operand > |
SHAML       < literal PE operand > |
SHAMR       < literal PE operand > |
SHAR        < literal PE operand > |
SKIP        < skip operand > |
SKIPF       < skip operand > |
SKIPFA      < skip operand > |
SKIPT       < skip operand > |
SKIPTA      < skip operand > |
SLIT        < ACAR selector >  < short literal operand > |
STA         < literal PE operand > |
STB         < literal PE operand > |
STL         < ACAR selector >  < CU memory operand > |
STORE       < ACAR selector >  < CU memory operand > |
STOREX      < ACAR selector >  < CU memory operand > |
STR         < literal PE operand > |
STS         < literal PE operand > |
STX         < literal PE operand > |
SUB         < PE address operand > |
SWAP        < blank PE operand > |
SWAPA       < blank PE operand > |
SWAPX       < blank PE operand > |
TCCW        < ACAR selector >  < blank CU operand > |
TCW         < ACAR selector >  < blank CU operand > |
TXEF        < ACAR selector >  < compare and skip operand > |
TXEFA       < ACAR selector >  < compare and skip operand > |
TXEFAM      < ACAR selector >  < skip operand > |
TXEFM       < ACAR selector >  < skip operand > |
TXET        < ACAR selector >  < compare and skip operand > |
TXETA       < ACAR selector >  < compare and skip operand > |
```

```
TXETAM      < ACAR selector >   < skip operand > |
TXETM       < ACAR selector >   < skip operand > |
TXGF        < ACAR selector >   < compare and skip operand > |
TXGFA       < ACAR selector >   < compare and skip operand > |
TXGFAM      < ACAR selector >   < skip operand > |
TXGFM       < ACAR selector >   < skip operand > |
TXGT        < ACAR selector >   < compare and skip operand > |
TXGRA       < ACAR selector >   < compare and skip operand > |
TXGTAM      < ACAR selector >   < skip operand > |
TXGTM       < ACAR selector >   < skip operand > |
TXLF        < ACAR selector >   < compare and skip operand > |
TXLFA       < ACAR selector >   < compare and skip operand > |
TXLFAM      < ACAR selector >   < skip operand > |
TXLFM       < ACAR selector >   < skip operand > |
TXLT        < ACAR selector >   < compare and skip operand > |
TXLTA       < ACAR selector >   < compare and skip operand > |
TXLTAM      < ACAR selector >   < skip operand > |
TXLTM       < ACAR selector >   < skip operand > |
WAIT        < blank CU operand > |
XD          < PE address operand > |
XI          < PE address operand > |
ZERF        < ACAR selector >   < skip operand > |
ZERFA       < ACAR selector >   < skip operand > |
ZERT        < ACAR selector >   < skip operand > |
ZERTA       < ACAR selector >   < skip operand > |
ZERXF       < ACAR selector >   < skip operand > |
ZERXFA      < ACAR selector >   < skip operand > |
ZERXT       < ACAR selector >   < skip operand > |
ZERXTA      < ACAR selector >   < skip operand > |
```

APPENDIX B

COMPLETE DESCRIPTION OF THE K-MACHINE

K-Machine Registers:

S Register. The S-Register indicates which location in the K-Machine stack is the top level. All binary operators use the top two operands in the stack, the top level for the right operand and the second level for the left operand.

I Register. The I Register holds one 32-bit ILLIAC IV instruction syllable. The instruction is built by calculating in the stack the values which define the fields of the instruction and storing them into their respective fields in the I Register.

LI Register. The LI Register holds the loader information for the instruction in the I Register. The LI may be stored into from the top of stack; and it is set automatically when a store into an address field of Register I is executed. When an instruction is emitted to the code file, the contents of Registers I and LI are joined together to form one 48-bit word in the code file.

ALLOCATIONCOUNTERS. These are the 64 allocation counters used by ASK.

ACN Register. ACN designates which one of the allocation counters (ALLOCATIONCOUNTERS) is in use.

AC Register. The AC Register holds the current value of the current allocation counter (ALLOCATIONCOUNTERS[ACN]).

L Register. The L Register indicates which K-Machine code syllable is currently being executed.

K Register. The K Register holds the K-Machine instruction currently being executed.

C Register. The C Register holds the record number of the last record in the card image file which was prepared for printing.

LN Register. The LN Register is the line image buffer for printing. It contains the next line to be printed.

PASS Register. The PASS Register holds the Pass number which the K-Machine is performing. PASS=0 implies Pass I, PASS=1 implies Pass II.

$\emptyset$ Register. The $\emptyset$ Register indicates the location of the next K-Machine instruction syllable to be written into the Pass II K-Machine input file, in the event that the K-Machine is copying instructions.

ABIT Register. The ABIT Register indicates a bit number in the top of stack. The bit is the first bit of a field whose width is given by the NBITS register.

BBIT Register. The BBIT Register indicates a bit number in the second level of the stack. The bit is the first bit of a field whose width is given by the NBITS Register.

NBITS Register. The NBITS Register gives the width of fields of bits in the top two locations in the stack.

GL Register. The GL Register gives the default Global/Local setting for ILLIAC IV CU instructions. Global means that all CUs synchronize their Instruction Counters before executing the instruction; local means that all CUs execute the instruction independently.

K-Machine Flip/Flops:

EFF. The execute flip/flop. If EFF=TRUE then K-Machine instructions will be executed in Pass I.

CFF. The copy flip/flop. If CFF=TRUE then the Pass I K-Machine will copy instruction syllables into the input file to the Pass II K-Machine.

LFF. The line buffer flip/flop. LFF=TRUE if Register LN holds a line image for printing.

FFF. The Fatal Error flip/flop. FFF=TRUE if the K-Machine has detected any errors during the executing of the K-Machine program.

SFF. The syntax flip/flop. SFF=TRUE if the SYNTAX option was ever used.

K-Machine Files:

File CARDS. File CARDS is the file addressed by Register C. It contains the saved card image input to ASK from Pass I. The format of a record of file CARDS is given in Section 4.6.4.

File KINPUT. This is the file addressed by Register L. It is the file containing the instruction syllables to be executed by the K-Machine.

File KOUTPUT. This is the file addressed by Register Ø during Pass I. It will be the same file that is used as KINPUT in Pass II.

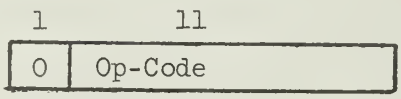
File CODE. File CODE contains the completely assembled ILLIAC IV instruction syllables.

File LINE. This is the line printer output file to which the listing is produced.

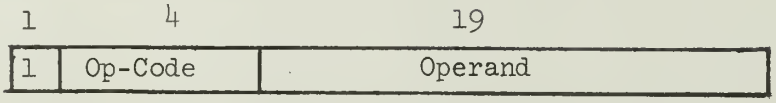
K-Machine Instruction Formats:

K-Machine instructions are either 12 or 24 bits in length. They are formatted as follows:

12 bit:



24 bit:



For 24-bit instructions, the operand field is interpreted according to the instruction itself. In the section which defines K-Machine operators, the operand fields of 24-bit instructions will be broken down individually for each operator.

K-Machine Operand Formats:

An operand in the K-Machine stack has the following format:

E. If field T is one then E is the external table address for this operand, otherwise E=0.

R. R gives the relocatability of this operand. R=0 means that it is Absolute

T. T indicates whether the operand is external or not. T=1 means that it is external and that E contains the external table address. T=0 means that it is either absolute or relocatable depending upon the value of R.

F. F indicates whether the stack location is an operand or a control word. F=0 means operand. No control words have as yet been found to be necessary in the K-Machine.

A. A indicates the type of arithmetic used to compute this operand, c.f. Section 1.2.1.

A=0 Row Arithmetic

A=1 Word Arithmetic

A=2 Syllable Arithmetic

VALUE. For arithmetic operations, only the low order 24 bits of VALUE are used. The remaining 8 bits are used for bringing instruction skeletons and data to the top of the stack. If T=1 then the VALUE field will hold the absolute displacement from the base determined by the external symbol.

K-Machine Operators:

The following set of K-Machine operators (12 and 24-bit) are executed by the K-Machine conditionally upon the logical value of the following Boolean expression (c.f. Section 4.6):

(PASSII means PASS=1)

PASSII OR (CFF IMP EFF)

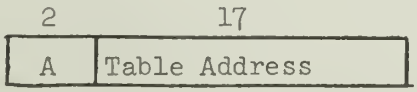
All K-Machine operators are subject to being copied into the Pass II input file depending upon the value of the following Boolean expression (c.f. Section 4.6):

(PASSI means PASS=0)

PASSI AND CFF

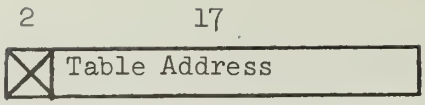
24-bit Operators:

OPDC Operand Call.



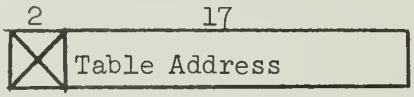
Fetch an operand from the symbol table. Adjust its value so that its arithmetic mode corresponds to that given in A (0 Row, 1 Word, 2 Syllable). Push the operand into the stack.

STD Store Destructive.



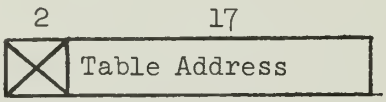
Store the top of stack into the indicated address in the symbol table Destroy the top of stack by reducing the S Register by one.

STN Store Non-Destructive.



Store the top of stack as in STD but do not adjust Register S.

FTCH Fetch from Symbol Table.



Bring the content of the indicated location to the top of stack. No adjustment is made for arithmetic mode as with OPDC.

12-bit Operators:

LIT1 One Syllable Literal.

Push the next syllable in the instruction stream into the stack as a 12-bit literal (high-order positions filled with zeroes).

LIT2 Two Syllable Literal.

Push the next two syllables into the stack as a 24-bit literal.

LIT3 Three Syllable Literal.

Push the next three syllables into the stack as a 36-bit literal.

LIT4 Four Syllable Literal.

Push the next four syllables into the stack as a 48-bit literal.

TAG Set Top 16 bits.

Set the high order 16 bits of the top of stack to the value given by the next two syllables in the instruction stream.

TAGD Tag Dynamic.

Set the high order 16 bits of the second level of the stack to the value given by the 16 low order bits of the top of stack. Reduce the S Register by one.

DIA Dial Top of Stack.

Set the ABIT Register to the value given by the next instruction syllable.

DIAD Dial Top of Stack Dynamic.

Set the ABIT Register to the value given in the top of stack. Reduce S by one.

DIB Dial Second Level of Stack.

Set the BBIT Register to the value given by the next instruction syllable.

DIBD Dial Second Level Dynamic.

Set the BBIT Register to the value found in the top of stack. Reduce S by one.

TRBITS Transfer Bits as Indicated by Registers.

Transfer the bits of the top of stack from the field defined by ABIT and NBITS to the field of the second level defined by BBIT and NBITS. Reduce S by one.

TRB Transfer Bits.

Set the NBITS Register to the value found in the next instruction syllable. Enter the TRBITS operator.

TRBD Transfer Bits Dynamic.

Set the NBITS Register to the value given in the top of stack. Enter the TRBITS operator.

DUP Duplicate the Top of Stack.

XCH Exchange.

Interchange the top two levels of the stack.

LC Loader Clear.

Clear the loader information register (LI).

Arithmetic Operators:

ADD Add Top Two Levels of Stack.

The top level of the stack is added to the second level of the stack. The result replaces the second level and the S Register is reduced by one.

Before the addition takes place, the T bits of the top two operands are examined. If both are 1, an error is flagged indicating that an attempt to add two External quantities has been detected.

The R fields of the operands are added together as well as the value field, the sum replacing the R field in the result.

The T fields are ORed together with the result replacing the T fields in the resulting operands.

The A field of the result is set to the larger of the two A fields of the operands.

MINUS Negate Top of Stack.

The value field and R field of the top of the stack are (Boolean) complemented and 1 added to the results (two's complement negation of the R field and VALUE field).

SUB Subtract the Top Two Levels of Stack.

Enter the MINUS operator. Enter the ADD operator.

If the second level of stack is Absolute or External and the top of stack is Relocatable, or if the top of stack is External, then an error is flagged indicating that the subtraction is invalid.

MUL Multiply Top Two Levels of Stack.

If both operands are not Absolute (R=0 and T=0) an error is flagged indicating that two non-absolute quantities were attempted to be multiplied together.

The second level of the stack is multiplied by the top of the stack. The result replaces the second level and S is reduced by one.

DIV Integer Divide.

If both operands are not Absolute (R=0 and T=0) an error is flagged indicating that a division involving two non-absolute quantities was attempted.

The second level of the stack is divided by the top of the stack. The division is performed using integer arithmetic. The result replaces the second level of the stack and S is reduced by one.

POWER Raise to the Power of.

If the two top levels are not both absolute then an error is flagged indicating an attempt to use non-absolute quantities in exponentiation.

The second level of the stack is raised to the power of the top of stack. The result replaces the second level and S is reduced by one.

Compare Operators:

GTR Greater Compare.

If the second level of the stack is greater than the top of the stack, the second level is replaced by an Absolute 1, otherwise 0. S is reduced by one.

GEQ Greater than or Equal to Compare.

If the second level of stack is greater than or equal to the top of stack, the second level is replaced by an Absolute 1, otherwise 0. S is reduced by one.

EQL Equal Compare.

If the second level of the stack is equal to the top of the stack, the second level is replaced by an Absolute 1, otherwise 0. S is reduced by one.

NEQ Not Equal Compare.

If the top two levels of the stack are not equal, then the second level is replaced by an Absolute 1, otherwise 0. S is reduced by one.

LEQ Less than or Equal Compare.

If the second level of the stack is less than or equal to the top of the stack, the second level is replaced by an Absolute 1, otherwise 0. S is reduced by one.

LSS Less Compare.

If the second level of the stack is less than the top of stack, the second level is replaced by an Absolute 1, otherwise 0. S is reduced by one.

Attribute Assignment Operators:

ABS Make Absolute.

Set the R field and T field of the top of the stack to zero.

REL Make Relocatable.

Set the R field of the top of the stack to 1; set the T field to zero.

RWA Row Arithmetic.

Set the A field of the top of the stack to zero.

WDA Word Arithmetic.

Set the A field of the top of stack to one.

SLA Syllable Arithmetic.

Set the A field of the top of the stack to two.

Logical Operators:

AND Logical And.

The top two levels of the stack are ANDed together. The result replaces the second level and S is reduced by one.

At least one of the operands must be absolute, if not an error is flagged.

The R field of the result is set to the larger R field value of the two operands. The T field of the result is set to the logical OR of the T

fields of the two operands. The A field of the result is set to the larger A field value of the two operands.

OR Logical Or.

The top two levels of the stack are ORed together. The result replaces the second level and S is reduced by one.

The setting of the R, T, and A fields of the result are determined as in the AND operator; likewise, at least one of the operands must be absolute an error being flagged if this is not the case.

EXOR Exclusive Or.

The top two levels of the stack are EXCLUSIVE-ORed together. The result replaces the second level and the S Register is reduced by one.

The setting of the R, T, and A fields of the result are determined as in the AND operator; likewise, at least one of the operands must be absolute, an error being flagged if this is not the case.

NOT Logical Negation.

The top of the stack is logically negated.

The top of stack operand must not be external else an error is flagged. The R, T, and A fields of the operand are not atlered.

Allocation Counter Operators:

SAC Store AC.

The AC Register is stored into ALLOCATIONCOUNTERS [ACN].

SACN Set ACN.

The ACN Register is set to the content of the top of stack operand. S is reduced by one. The AC Register is set from the content of ALLOCATION-COUNTERS [ACN].

BAC Bump AC.

The VALUE field of the AC Register is increased by one.

OAC Orgin AC.

The top of stack operand replaces the AC Register; S is reduced by one.

AAC Add to AC.

The AC Register is placed in the top of stack. The ADD operator is entered. The result replaces the AC Register; S is reduced by one.

LAC Load AC.

The AC Register is placed in the top of stack.

Branch Operators:

BB Branch Backward.

The L Register is decreased by the content of the top of stack. The S Register is reduced by one.

BF Branch Forward.

The L Register is increased by the content of the top of stack. The S Register is reduced by one.

BBC Branch Backward Conditional.

If the low order bit of the second level of the stack is a 0 then the L Register is decreased by the content of the top of stack. The S Register is reduced by two.

BFC Branch Forward Conditional.

If the low order bit of the second level of the stack is a 0 then the L Register is increased by the content of the top of stack. The S Register is reduced by two.

ICX Instruction Store ACARX Field.

The three low order bits of the top of stack replace the entire ACARX field of the I Register. The S Register is reduced by one.

If the top of stack operand is not Absolute, an error is flagged.

IACV Instruction Store ACAR Field.

The low order two bits of the top of stack are stored into the ACAR field of the I Register.

If the top of stack operand is not Absolute, an error is flagged.

SKIP Instruction Store Skip Field.

If the top of stack operand is relocatable then the top of stack is made Absolute, the VALUE field of the AC Register is placed on top of the stack, and the following operators entered: XCH, SUB.

The top of stack operand is now examined to see if it is negative. If it is negative then a 1 is stored in the sign bit of the Skip field of the I Register and the top of stack in negated (two's complement), otherwise a 0 is stored into the sign bit of the Skip field of the I Register.

If the top of stack operand is greater than 127, an error is flagged.

The low order seven bits of the top of stack are stored into the low order seven bits of the Skip field of the I Register. The S Register is reduced by one.

IPU Instruction Store ADR Use Field.

The top of the stack is stored into the ADR Use field of the I Register. The S Register is reduced by one.

If the top of stack operand is not Absolute, an error is flagged.

IPA Instruction Store PE Address Field.

If the R field and the T field of the stack operand are both non-zero, an error is flagged.

The low order 16 bits of the top of stack are stored into the PE address field of the I Register. The LI register is set according to the relocatability, externalness and type of arithmetic value of the top of stack operand. S is reduced by one.

IPN Instruction Store N Field.

If the top of stack operand is not Absolute, an error is flagged.

The low order eight bits of the top of stack are stored into the low order eight bits of the I Register. S is reduced by one.

IPR Instruction Store R Field.

If the top of stack operand is not Absolute, an error is flagged.

The low order bits of the top of stack are stored into the Routing Register field of the I Register. S is reduced by one.

ICA Instruction Store CU Address Field.

If the top of stack operand is not Absolute, an error is flagged.

The low order eight bits of the top of stack are stored into the CU address field of the I Register. The S Register is reduced by one.

ICAC Instruction Store CU Address Field and Check.

If the second level of stack operand is not Absolute, an error is flagged.

The low order eight bits of the second level of the stack are tested according to a bit mask found in the top of stack. The mask has one bit corresponding to each register (or set of registers) in the ILLIAC IV CU.

If the bit is on, then that register is legally addressable; if it is off, then it is not legally addressable. If the address in the second level of the stack corresponds to an off-bit in the top of stack, an error is flagged.

The second level of the stack is stored into the CU Address field of the I Register. The S Register is reduced by two.

ISAJ Instruction Store Slit/Alit/Jump.

If the R field and the T field of the top of stack operand are both non-zero, an error is flagged.

The low order 24 bits of the top of stack operand are stored into the low order 24 bits of the I Register. The LI register is set according to the relocatability, externalness and type of arithmetic value of the top of stack operand. The S Register is reduced by one.

IGL Instruction Store Global/Local.

The GL register is stored into the Global/Local field of the I Register.

IGLB Instruction Store Global.

The Global/Local field of the I Register is set to 0 (Global).

ILCL Instruction Store Local.

The Global/Local field of the I Register is set to 1 (Local).

GLBL Global.

The GL Register is set to 0 (Global).

LCL Local.

The GL Register is set to 1 (Local).

Miscellaneous Operators:

PRNT Print.

The PRNT operator is explained in some detail in Section 4.6.4. The value "n" (c.f. section 4.6.4) is obtained from the next two K-Machine instruction syllables in the input stream.

EMT Emit.

Emit one 32-bit syllable ILLIAC IV instruction into the object code file. The syllable is formed by concatenating the low order 16 bits of the LI Register with the low order 32 bits of the I Register. No instructions are written into the object code file if either FFF (Fatal error Flip/Flop) or SFF (Syntax Flip/Flop) is one.

SSFF Set Syntax Flip/Flop.

The SFF is set to TRUE.

Instructions not Affected by CFF or EFF:

EXE Execute Enable.

Set EFF to TRUE.

EXD Execute Disable.

Set EFF to FALSE.

CPYE Copy Enable.

Set CFF to TRUE.

CPYD Copy Disable.

Set CFF to FALSE.

EXIT Exit the K-Machine.

The K-Machine ceases to execute instructions.

LIST OF REFERENCES

[1] Alsberg, P. A., Gaffney, J. L., Grossman, C. R., Mason, T. W., and Westlund, G. A., "A Description of the ILLIAC IV Operating System", ILLIAC IV Document No. 212, Department of Computer Science, University of Illinois, Urbana, Illinois, (March 1969).

[2] Grothe, D. M., Luskin, C., "Reference Manual for ILLIAC IV Assembler (ASK)' Burroughs Corporation Document No. 66072, (March, 1969).

[3] "ILLIAC IV Systems Characteristics and Programming Manual", Burroughs Corporation Document No. 66000A, (June, 1969).

[4] Strachey, C., "A General Purpose Macrogenerator", The Computer Journal, Vol. 8, No. 3 (October 1965), p. 225.

[5] Barnes, G., et. al., "The ILLIAC IV Computer", IEEE Trans. on Computers, Vol. C-17 (August 1968), p. 746.

LASSIFIED

Security Classification

DOCUMENT CONTROL DATA - R & D

(Security classification of title, body of abstract and indexing annotation must be entered when the overall report is classified)

GINATING ACTIVITY *(Corporate author)*

epartment of Computer Science
iiversity of Illinois at Urbana-Champaign
rbana, Illinois 61801

2a. REPORT SECURITY CLASSIFICATION

UNCLASSIFIED

2b. GROUP

ORT TITLE

MACRO-ASSEMBLER FOR ILLIAC IV

CRIPTIVE NOTES *(Type of report and inclusive dates)*

search Report

THOR(S) *(First name, middle initial, last name)*

vid Michael Grothe

ORT DATE	7a. TOTAL NO. OF PAGES	7b. NO. OF REFS
cember 1, 1969	114	5

NTRACT OR GRANT NO.	9a. ORIGINATOR'S REPORT NUMBER(S)
S AF 30(602)4144 OJECT NO. 6-26-15-305	DCS Report No. 364
	9b. OTHER REPORT NO(S) *(Any other numbers that may be assigned this report)*

STRIBUTION STATEMENT

ualified requesters may obtain copies from DCS.

PPLEMENTARY NOTES	12. SPONSORING MILITARY ACTIVITY
ne	Rome Air Development Center Griffiss Air Force Base Rome, New York 13440

STRACT

This report describes the ILLIAC IV macro assembly language (ASK) and the ILLIAC IV macro assembler. ASK is a free field assembly language with conditional assembly features and in-line text-substitution macros.

D FORM 1 NOV 65 1473

UNCLASSIFIED

Security Classification

14. KEY WORDS	LINK A		LINK B		LINK C	
	ROLE	WT	ROLE	WT	ROLE	WT
ILLIAC IV						
assembler						
macro assembler						
conditional assembly						